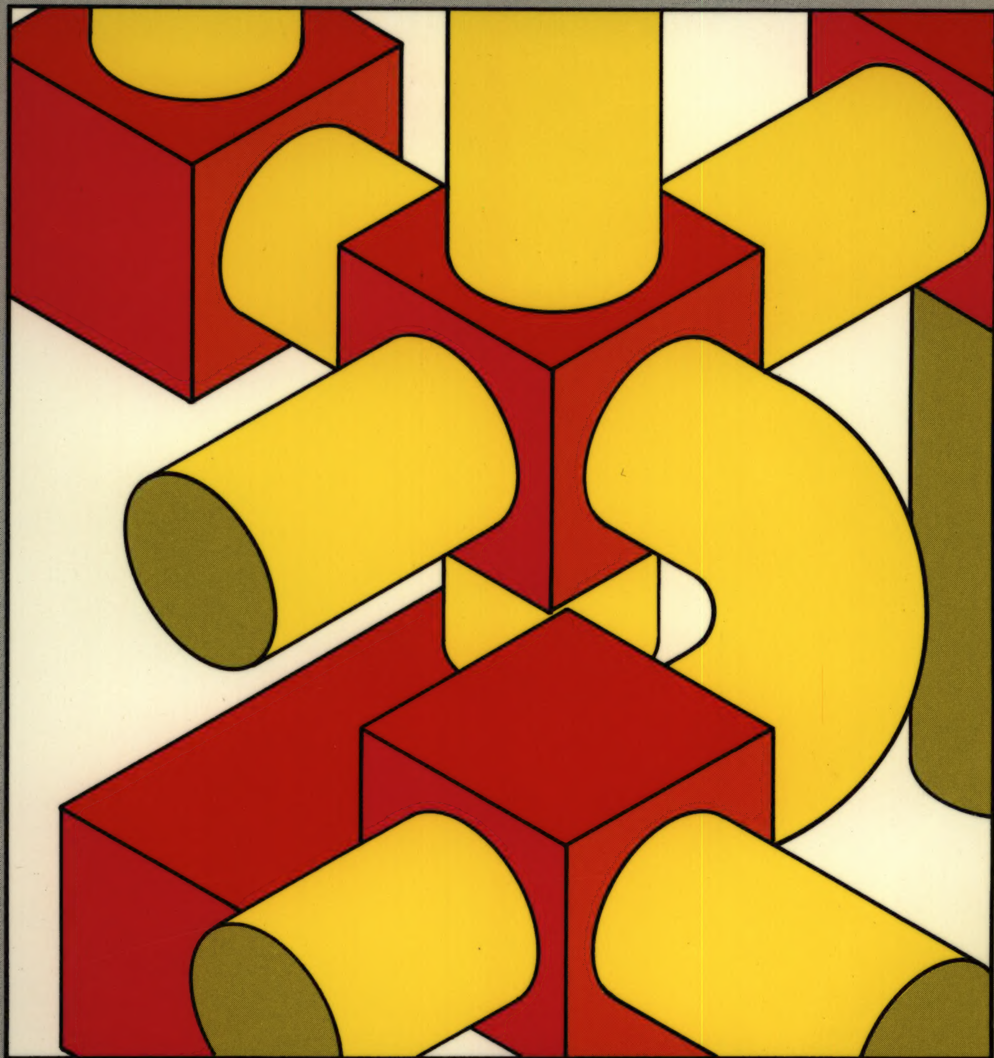
MODULA-2

E. VERHULST



ACADEMIC SERVICE

MODULA-2

MODULA-2

E. VERHULST

ACADEMIC SERVICE

CIP-GEGEVENS KONINKLIJKE BIBLIOTHEEK, DEN HAAG

Verhulst, E.

Modula-2 / E. Verhulst. - Den Haag : Academic Service
Met lit. opg., reg.
ISBN 90-6233-212-9
SISO 365.3 SVS 8.12.3 UDC 800.92 Modula-2 UGI 200
Trefw.: Modula-2 (programmeertaal).

Uitgegeven door: Academic Service
Postbus 81
2870 AB Schoonhoven
Zetwerk: E. Verhulst
Omslagontwerp: JAM Gauw
Druk: Krips Repro Meppel
Bindwerk: Meeuwis, Amsterdam
ISBN 90 6233 212 9

Inhoudsopgave

Hoofdstuk 1 : Inleiding 1
1.1 De geschiedenis van Modula-2 1
1.2 Eigenschappen van Modula-2 2
1.3 Enkele voorbeelden 8
1.4 Abstractieschema 15
1.5 Verborgen typen 19
1.6 Een Modula-2 ontwikkelingsomgeving 24
1.6.1 De compiler 25
1.6.2 Versiecontrole 29
1.6.3 De symbolische debugger 31

Hoofdstuk 2 : Modula-2 woordenschat 35
2.1 Syntaxisnotatie 35
2.2 Woordenschat 37

Hoofdstuk 3 : Elementaire programma-objecten 49
3.1 Enkelvoudige typen 49
3.2 Het declareren van identifier 64
3.3 Declaratie van programma-objecten 73
3.4 Operanden met overdraagbaar type 74
3.5 De toekenningsopdracht 75

Hoofdstuk 4 : Besturingsstructuren 81
4.1 HALT en RETURN 81
4.2 De sequentiële structuur of de opdrachtenrij 82
4.3 Conditionele structuren 83
4.4 Herhalingsopdrachten 90
4.5 De vorm van een programmamodule 103

Hoofdstuk 5 : Samengestelde typen 111
5.1 Het tabeltype ARRAY 111
5.2 Het recordtype RECORD 123
5.3 Het verzamelingtype SET 131
5.4 Het standaardtype BITSET 137

Hoofdstuk 6 : Procedures 151
6.1 De proceduredeclaratie 151
6.2 De procedure-aanroep 156

6.3 Open-array-parameters 157
6.4 Het bereik van objecten 159
6.5 De functieprocedure 161
6.6 De declaratie van een functieprocedure 162
6.7 De functie-aanroep 165
6.8 Neveneffecten 165
6.9 Het proceduretype en de procedurevariabele 166
6.10 Toepassingen van procedurevariabelen 168

Hoofdstuk 7 : Recursie 175
7.1 Iteratie en recursie 175
7.2 Indirecte recursie 187

Hoofdstuk 8 : Wijzers en dynamische variabelen 191
8.1 Het wijzertype en de wijzervariabele 191
8.2 Het creëren en vrijgeven van naamloze variabelen 193
8.3 De implementatie van NEW en DISPOSE 194
8.4 Het toewijzen en vrijgeven van records met varianten 196
8.5 Toepassingen 198

Hoofdstuk 9 : Bibliotheekmodulen 211
9.1 Modulen compileren 212
9.2 In- en uitvoer van objecten 213
9.3 Definitiemodule 217
9.4 Implementatiemodule 221
9.5 Het initiëren van variabelen in een implementatiemodule 229
9.6 De levensduur van variabelen in een implementatiemodule 231
9.7 De volgorde bij het compileren 232
9.8 Het ontwerp van bibliotheekmodulen 234
9.9 Het gebruik van bibliotheekmodulen 238

Hoofdstuk 10 : Interne modulen 243
10.1 De in- en uitvoer van objecten 243
10.2 Het bereik van identifiers 244
10.3 Impliciete in- en uitvoer 252

Hoofdstuk 11 : De Modula-2 standaardbibliotheek 261

Hoofdstuk 12 : Strings 265
12.1 Inleiding 265
12.2 Stringconstanten 265
12.3 Strings met variabele lengte 266
12.4 Bewerkingen met strings 267
12.5 Toepassingen 276
12.5.1 Anagramprogramma 276
12.5.2 Getalrepresentatie als string 285
12.5.3 Programma Fibonacci 292

Hoofdstuk 13 : Conversie 295
13.1 De module Convert 295
13.2 De module ConvertReal 297
13.3 Rekenen met getallen in een stuk tekst 298

Hoofdstuk 14 : Wiskundige functies 303

Hoofdstuk 15 : Het werkstation 305
15.1 Module Terminal 305
15.2 Een uitbreiding voor de module Terminal 308

Hoofdstuk 16 : Werken met bestanden 315
16.1 Inleiding 315
16.2 Fatale en niet-fatale fouten 315
16.3 De module Files 316
16.4 De toegang tot een bestand 317
16.5 De toestand van een bestand 318
16.6 Het openen van een bestand 320
16.7 Het afsluiten van een bestand 321
16.8 Het instellen van een bestand 321
16.9 De toestand en de naam van een bestand 322
16.10 De module Binary 324
16.11 De module FilePositions 326

Hoofdstuk 17 : Tekstbestanden 331
17.1 De module Text 331
17.2 De leesopdrachten 332
17.3 De schrijfopdrachten 334
17.4 De module NumberIO 338

Hoofdstuk 18 : De standaardin- en -uitvoer 343
18.1 De module SimpleIO 343
18.2 De module RealIO 344
18.3 De module StandardIO 345

Hoofdstuk 19 : Beheer van een inhoudstabel 351
19.1 Module Directory 351

Hoofdstuk 20 : De aanroep van subprogramma's 353

Hoofdstuk 21 : Het geheugenbeheer 363
21.1 De module Storage 363

Appendix : Syntaxisdiagrammen 365

Index 385

Voorwoord

Dit boek is een volledige behandeling van de programmeertaal Modula-2. Modula-2 is in 1978 ontworpen door N. Wirth als een uitbreiding van Pascal. Deze uitbreiding is enerzijds naar een hoger niveau, van de machine af : de taal ondersteunt de moderne, object georiënteerde programmatuurontwikkeling. De uitbreiding is echter ook naar de machine toe, met mogelijkheden op machine-niveau.

In het boek onderscheiden we drie belangrijke delen : het eerste deel, hoofdstuk 1, beschrijft enkele belangrijke eigenschappen van Modula-2. Ook wordt een techniek voorgesteld voor het schematisch weergeven van de module-afhankelijkheid in een programma. Het deel eindigt met de beschrijving van een Modula-2 ontwikkelingsomgeving. Enkele belangrijke hulpmiddelen voor het opbouwen van een Modula-2 programmatuursysteem worden toegelicht. Dit hoofdstuk kan dienen als een leidraad en een hulpmiddel bij het kennismaken en het leren gebruiken van de Modula-2 implementatie waarover de lezer kan beschikken. Het tweede deel, hoofdstukken 2 tot 10, behandelt de syntaxis en de semantiek van de programmeertaal. Vooral in de hoofdstukken 9 en 10 wordt de nadruk gelegd op het ontwerpen en gebruiken van modulen. Het laatste deel, hoofdstukken 11 tot 20, is een gedetailleerde beschrijving van een als standaard voorgestelde modulebibliotheek. Met deze bibliotheek streven we de overdraagbaarheid van Modula-2 programma's na tussen verschillende computersystemen of zelfs besturingssystemen. Het boek eindigt met twee bijlagen : een overzicht van syntaxisdiagrammen en een uitgebreide index.

Elk hoofdstuk wordt afgesloten met een reeks zorgvuldig gekozen oefenopgaven en een literatuurlijst voor verdere studie. De oplossingen van deze oefeningen en de broncode van de standaardbibliotheek zijn beschikbaar op een afzonderlijke PC-DOS overdraagbare diskette. Voor de lesgevers is ook een set transparanten met de figuren en de voorbeelden uit de tekst te verkrijgen.

De programmeertaal Modula-2 en vooral de standaardbibliotheek zijn nog volop in ontwikkeling. Dit boek is dan ook slechts een momentopname. Ook worden in dit boek niet alle nieuwe mogelijkheden behandeld; hiervoor wordt verwezen naar een tweede deel. Voor correcties en suggesties houd ik me aanbevolen.

Graag wil ik mijn dank uitspreken voor de hulp en de talrijke kritische opmerkingen die ik kreeg van de heren L. Bux, R. De Wachter, F. Haes, L. Rossi en P. Schavey en voor de hulp van C. Van Loon bij het tekenen van de figuren.

Deurne, mei 1986.

1 Inleiding

1.1 De geschiedenis van Modula-2

Leren programmeren steunt op het begrijpen van elementaire concepten zoals data- en besturingsstructuren. Hiervoor gebruiken we bij voorkeur een notatie die deze concepten duidelijk weergeeft en het gebruik van geordende structuren beklemtoont. Met dit doel heeft prof. Niklaus Wirth in 1968 de programmeertaal Pascal gedefinieerd. De implementatie van Pascal-compilers voor micro-computers heeft deze taal zeer populair gemaakt in het onderwijs en voor het programmeren van kleinere projecten.

In 1978 start prof. Niklaus Wirth met het ontwerp van het individuele werkstation Lilith. Dit werkstation heeft onder meer de volgende eigenschappen : een bitmap-beeldscherm (met elk beeldpunt op het scherm komt een bit in het geheugen overeen), een muis als invoerapparaat en een architectuur die is aangepast aan het besturingssysteem. Bij het ontwerp van de Lilith-computer is vastgelegd dat slechts één enkele programmeertaal mag worden gebruikt. Deze taal moet geschikt zijn voor de beschrijving van algoritmen op een hoog abstractieniveau, voor het weergeven van bewerkingen op machineniveau, voor de implementatie van gegevens-banken en verder voor alle systeemprogramma's. Pascal is voor dit doel niet geschikt. Ook een Pascal-implementatie met uitbreidingen is niet als een oplossing aanvaard. Steunend op de principes van Pascal en talen geschikt voor multiprogrammering, heeft prof. Niklaus Wirth de taal Modula-2 gedefinieerd.

We kunnen Modula-2 beschouwen als de logische opvolger van Pascal en een volwaardig alternatief voor de complexe programmeer-taal Ada, de nieuwe taal van het ministerie van defensie van de Verenigde Staten. Modula-2 weerspiegelt een decennium ervaring met Pascal : sommige onvolkomenheden zijn weggewerkt en de taal bevat talrijke verbeteringen zodat de Pascal-dialecten overbodig worden. De belangrijkste nieuwe eigenschap van Modula-2 is de ver doorgevoerde ondersteuning voor het programmeren met modulen.

1.2 Eigenschappen van Modula-2

Leesbaarheid

Programma's worden veel vaker gelezen dan geschreven. De leesbaarheid wordt verbeterd indien in een taal talrijke gegevens- en besturingsstructuren aanwezig zijn. Modula-2 bevat alle structuren die reeds in Pascal aanwezig zijn. Sommige van deze structuren zijn verbeterd :

Pascal	Modula-2
- vaste tabellen;	- open tabellen;
- strikte volgorde voor de declaraties;	- vrije volgorde voor de declaraties;
- CASE-opdracht zonder ELSE-clausule;	- CASE-opdracht met een ELSE-clausule;
- volledige evaluatie van samengestelde logische uitdrukkingen;	- voorwaardelijke evaluatie van samengestelde logische uitdrukkingen;
- samengestelde opdrachten BEGIN ... END;	- gestructureerde opdrachten die alle eindigen met END, zoals WHILE ... DO ... END;
- FOR-opdracht met een stap +1 of -1;	- FOR-opdracht met een BY-clausule;
- labels en GOTO-opdrachten;	- besturingsopdrachten RETURN, EXIT en HALT.

Modula-2 onderscheidt zowel grote als kleine letters in de namen van de objecten die in een programma worden gebruikt. De gereserveerde woorden en standaardidentifiers worden steeds met hoofdletters geschreven.

Declaratie van objecten

Elk object, bijvoorbeeld een constante, een variabele, een type, een procedure, moeten we vooraf declareren. De formele definities van de objecten, eventueel aangevuld met commentaar, verbeteren de leesbaarheid en het onderhoud van programma's aanzienlijk. De declaraties verschaffen ook informatie aan het vertaalprogramma voor de reservering van geheugenruimte. De volgorde waarin we in Modula-2 objecten declareren is vrijer dan de strikte volgorde die vereist is in Pascal-programma's.

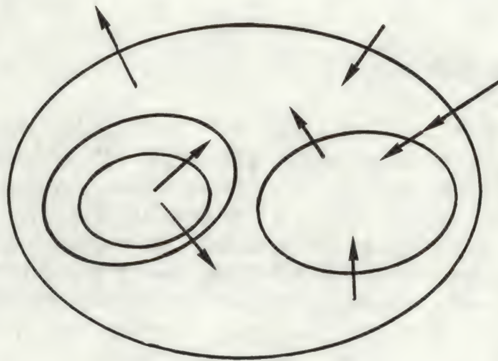
Typering van de variabelen

Elke variabele is van een bepaald type. Het type van de variabele bepaalt de operaties op die variabele. Tijdens de vertaling controleert het vertaalprogramma de bewerkingen uitgevoerd op de variabelen. Talrijke fouten, zoals uitdrukkingen waarin variabelen van een verschillend type met elkaar worden vermengd, worden reeds bij de vertaling ontdekt. Modula-2 bevat de basistypen van Pascal : INTEGER, REAL, BOOLEAN, CHAR. Voor de bewerkingen met natuurlijke getallen is het type CARDINAL toegevoegd. Het type BITSET maakt de manipulatie van bitpatronen voor het programmeren van laagniveaufuncties mogelijk. De module SYSTEM bevat daarnaast nog een aantal basistypen voor het programmeren van systeemafhankelijke modulen : BYTE, WORD en ADDRESS.

Met behulp van deze basistypen kunnen we nieuwe gestructureerde typen definiëren. De basisstructuren zijn het deelintervaltype, de ARRAY, de RECORD, de SET en de POINTER.

Grote programma's

De MODULE is een mechanisme voor het nesten van een aantal programma-objecten. Met een module ontstaat een nieuw niveau voor de opsplitsing van programma's. De modulen kunnen door verschillende programmeurs en eventueel gespreid in de tijd worden opgebouwd. De zichtbaarheid van de objecten in een module voor andere modulen wordt geregeld door de IMPORT- en EXPORT-functies.



figuur 1.1 IMPORT en EXPORT van objecten tussen modulen

De koppeling van een module met andere modulen moet steeds worden overwogen voordat we pogen de module te implementeren. Met Modula-2 kunnen we de moduledefinitie of de koppeling met andere

modulen afzonderlijk van de eigenlijke implementatie definiëren. Een module is opgebouwd uit een DEFINITION MODULE en een IMPLEMENTATION MODULE.

Abstractie

Abstractie is een essentieel hulpmiddel bij het beheersen van complexiteit. Abstractie maakt het mogelijk aspecten van een complex systeem afzonderlijk te behandelen. Essentieel is de scheiding tussen het wat (DEFINITION MODULE) en het hoe (IMPLEMENTATION MODULE).

Een abstract datatype is een datastructuur met daaraan gekoppeld een aantal procedures om bewerkingen op een waarde van die datastructuur te kunnen uitoefenen. Deze bewerkingen zijn bedoeld om gegevens op te vragen over de in de datastructuur opgeslagen waarden of om de mogelijkheid te bieden die waarden op een gecontroleerde manier te wijzigen. De door de procedures gegeven bewerkingen zijn de enige bewerkingen die op zo een datastructuur kunnen worden uitgevoerd.

Een Modula-2 module herbergt meestal een verzameling gegevensobjecten en biedt over het algemeen een verzameling bewerkingen om deze gegevens te manipuleren. De toegang van een gebruiker van een module geschiedt dus steeds via de geboden procedures. Dit biedt belangrijke voordelen : de consistentie van de gegevensobjecten wordt uitsluitend in de module gecontroleerd en het is steeds mogelijk de bewerkingen op andere wijzen te implementeren. Met Modula-2 kunnen abstracte datatypen zonder moeilijkheden worden gerealiseerd.

Afzonderlijke compilatie

De verschillende modulen, ook het definitie- en het implementatiedeel van een module, kunnen afzonderlijk worden gecompileerd. Tijdens de compilatie worden de eigenschappen van de objecten, waaraan in verschillende modulen kruisgewijs wordt gerefereerd, intens gecontroleerd. Alle eigenschappen van een object, dat uit een andere module via een IMPORT-functie in de gebruikersmodule zichtbaar is gemaakt, worden gecontroleerd bij de uitvoering van bewerkingen.

Bibliotheek van modulen

Met de modulen bouwen we bibliotheken op. In elke Modula-2 implementatie zijn werktuigen aanwezig voor het beheer van deze modulebibliotheek. Een belangrijke eigenschap is de automatische versiecontrole van modulen.

Elke Modula-2-implementatie bevat een bibliotheek met modulen voor de uitvoering van elementaire bewerkingen. Deze bibliotheek is nog niet internationaal gestandaardiseerd. Gewoonlijk bestaat een bibliotheek uit de volgende modulen :

- in- en uitvoer via het werkstation;
- conversie van getallen van tekst naar binaire voorstelling en omgekeerd;
- bewerkingen met bestanden : read, write en seek;
- bewerkingen met inhoudstabellen voor bestanden : create, delete en rename;
- dynamisch geheugenbeheer : new en dispose;
- uitvoering van programma's;
- planning van processen;
- wiskundige functies;
- dynamisch beheer van strings;
- systeemafhankelijke objecten.

De bibliotheek bevat naast deze modulen ook alle andere modulen die we zelf definiëren of aan het systeem toevoegen. Een programmatuursysteem kunnen we opbouwen zoals een kind speelt met een blokkendoos : modulen van een hoog niveau worden samengesteld uit modulen van een lager niveau, grote programma's worden opgebouwd met kleinere programma's. Op het laagste niveau vinden we de 'standaardmodulen'. Deze modulen zorgen voor de verbinding met het besturingssysteem.

De 'standaardmodule' SimpleIO

De module SimpleIO bevat de elementaire bewerkingen voor de standaardin- en -uitvoer ('input' en 'output'). Deze module is voldoende voor het programmeren van problemen waarbij de communicatie via de standaardin- en -uitvoer geschiedt.

```
DEFINITION MODULE SimpleIO;
FROM SYSTEM IMPORT WORD;
EXPORT QUALIFIED
  (* proc *) EOL, EOT,
             ReadChar, ReadString, ReadLn,
             ReadInt, ReadCard, ReadNum,
             CondRead, UndoRead,
             WriteChar, WriteLn, WriteString,
             WriteInt, WriteCard, WriteNum;

PROCEDURE EOT() : BOOLEAN;
(* EOT levert de waarde TRUE bij een leesopdracht na het einde
   van de invoerstroom; anders FALSE.
*)
```

```
PROCEDURE EOL() : BOOLEAN;
(* EOL levert de waarde TRUE bij een leesopdracht na het einde
   van een regel van de invoerstroom; anders FALSE.
*)

PROCEDURE ReadChar(VAR ch : CHAR);
(* ReadChar leest een teken van de standaardinvoer 'input'.
*)

PROCEDURE ReadString(VAR s : ARRAY OF CHAR);
(* ReadString leest een rij tekens.  De leesopdracht eindigt als
   - EOL of EOT TRUE wordt of
   - de variabele s is volledig gevuld.
*)

PROCEDURE ReadLn;
(* ReadLn leest de tekens tot juist na het einde van de regel.
*)

PROCEDURE ReadInt(VAR x      : INTEGER;
                  VAR succes : BOOLEAN);
(* ReadInt leest een string en vormt deze om tot een integer;

   syntaxis : ['+' | '-'] cijfer {cijfer}

   Leidende voorspaties worden genegeerd.
   succes := 'integer is gelezen'
*)

PROCEDURE ReadCard(VAR x      : CARDINAL;
                   VAR succes : BOOLEAN);
(* ReadCard leest een string en vormt deze om tot een cardinal;

   syntaxis : cijfer {cijfer}

   Leidende voorspaties worden genegeerd.
   succes := 'cardinal is gelezen'
*)

PROCEDURE ReadNum(VAR x      : WORD;
                      basis  : CARDINAL;
                  VAR succes : BOOLEAN);
(* ReadNum leest een string en vormt deze om tot een cardinal of
   een integer met de opgegeven basis.  Voor de basis geldt :
   2 <= basis <= 36;
   syntaxis : cijfer {cijfer}
```

```
   Leidende voorspaties worden genegeerd.
   succes := 'getal is gelezen'
*)

PROCEDURE CondRead(VAR ch     : CHAR;
                   VAR succes : BOOLEAN);
(* CondRead leest een teken van de standaardinvoer 'input'.  Als
   echter geen teken wordt ingevoerd wacht CondRead niet en levert
   voor succes de waarde FALSE.  Als een teken is ingevoerd wordt
   succes TRUE.
*)

PROCEDURE UndoRead ();
(* UndoRead plaatst het laatste ingevoerde teken terug in de
   invoerstroom.
*)

PROCEDURE WriteChar(      ch : CHAR);
(* WriteChar schrijft een teken naar de standaarduitvoer 'output'.
*)

PROCEDURE WriteLn;
(* WriteLn schrijft het teken 'einde regel' naar de standaarduit-
   voer 'output'.
*)

PROCEDURE WriteString(      s : ARRAY OF CHAR);
(* WriteString schrijft een string naar de standaarduitvoer
   'output'.
*)

PROCEDURE WriteInt(      x : INTEGER;
                         n : CARDINAL);
(* WriteInt vormt de integer x om tot een string en schrijft de
   string naar de standaarduitvoer 'output'.
   De string bestaat uit ten minste n tekens.  Indien n groter is
   dan het aantal tekens nodig voor x, worden leidende voorspaties
   toegevoegd.  Indien n onvoldoende is om de juiste waarde van x
   weer te geven worden automatisch extra tekens toegevoegd.
*)

PROCEDURE WriteCard(      x : CARDINAL;
                          n : CARDINAL);
(* WriteCard vormt cardinal x om tot een string en schrijft de
   string naar de standaarduitvoer 'output'.
   De string bestaat uit ten minste n tekens.  Indien n groter is
   dan het aantal tekens nodig voor x, worden leidende voorspaties
   toegevoegd.  Indien n onvoldoende is om de juiste waarde van x
   weer te geven worden automatisch extra tekens toegevoegd.    *)
```

```
PROCEDURE WriteNum(      x     : WORD;
                        basis : CARDINAL;
                        n     : CARDINAL);
(* WriteNum vormt x om tot een string tekens in de gegeven basis
   en schrijft de string naar de standaarduitvoer 'output'.
   De string bestaat uit ten minste n tekens.
   Indien n groter is dan het aantal tekens nodig voor x, worden
   leidende voorspaties toegevoegd.  Indien n onvoldoende is om
   de juiste waarde van x weer te geven worden automatisch extra
   tekens toegevoegd.
*)

END SimpleIO.
```

1.3 Enkele voorbeelden

De grootste gemene deler

Beschouw het volgende probleem :

De invoer van een programma bestaat uit twee natuurlijke getallen groter dan nul. De uitvoer bestaat uit de twee gelezen getallen en de grootste gemene deler van deze getallen.

Voor de oplossing van dit probleem gebruiken we de volgende eigenschappen uit de wiskunde :

1. als x = y, dan is x of y de grootste gemene deler ggd van x en y;

2. de ggd van twee getallen wijzigt niet, als we het grootste van beide getallen vervangen door het verschil van het grootste en het kleinste.

We geven de formulering van het programma voor dit probleem in Pascal en in Modula-2.

Pascal-programma :

```
program ggd(input,output);
var x,y,a,b : 1..maxint;
```

```
begin
write('Geef een eerste positief getal : ');
readln(a);
write('Geef een tweede positief getal : ');
readln(b);
x := a;
y := b;

while x <> y do
  if x > y
  then x := x - y
  else y := y - x;

writeln('de ggd van ',a : 3, ' en van ',b : 3, ' is ',x : 3)

end {ggd}.
```

Modula-2 programma :

```
MODULE Ggd;
FROM SimpleIO IMPORT ReadCard, ReadLn,
                     WriteString, WriteLn, WriteCard;

VAR x, y   : CARDINAL;
    a, b   : CARDINAL;
    succes : BOOLEAN;

BEGIN
WriteString('Geef een eerste positief getal :');
ReadCard(a,succes);
ReadLn;
WriteString('Geef een tweede positief getal :');
ReadCard(b,succes);
ReadLn;
x := a;
y := b;

WHILE x # y DO
  IF x > y
  THEN x := x - y
  ELSE y := y - x
  END
END;

WriteString('de ggd van ');
WriteCard(a,3);
WriteString(' en van ');
```

```
WriteCard(b,3);
WriteString(' is ');
WriteCard(x,3);
WriteLn

END Ggd.
```

In Modula-2 bestaat een programma uit een programmamodule. Deze module is opgebouwd zoals een Pascal-programma. De programmamodule begint met

MODULE <naam>

en eindigt met

END <naam>.

In de programmamodule vermelden we alle bibliotheekmodulen waarvan objecten in de programmamodule worden gebruikt. Deze objecten worden met de naam van de bibliotheekmodule in een invoerlijst vermeld. Daarna volgen de declaraties van de objecten die in het programma worden gebruikt. Het actiedeel is, op enkele syntactische verschillen na, hetzelfde als het actiedeel van een Pascal-programma. De belangrijkste verschillen vinden we terug bij de in- en uitvoer : voor elk type variabele is in de bibliotheekmodule een afzonderlijke procedure gedefinieerd. De Pascal-opdracht

```
writeln('de ggd van ',a : 3,' en van ',b : 3,' is ',x : 3)
```

coderen we in Modula-2 met de afzonderlijke opdrachten :

```
WriteString('de ggd van ');
WriteCard(a, 3);
WriteString(' en van ');

WriteCard(b, 3);
WriteString(' is ');
WriteCard(x, 3);
WriteLn
```

Het omkeren van een reeks tekens

Beschouw eens het volgende probleem. De invoer van een programma is een reeks tekens. De tekens worden ingevoerd via het toetsenbord van het werkstation. De invoer wordt beëindigd met de aanslag van de RETURN- of ENTER- toets. De uitvoer van het programma is de reeks tekens in omgekeerde volgorde. De tekens worden weergegeven op het beeldscherm van het werkstation.

Voor de oplossing van dit probleem kunnen we drie stappen onderscheiden :
- het lezen van de reeks tekens;
- het omkeren van de reeks;
- het weergeven van de omgekeerde reeks.

Elk van deze drie deelproblemen kan op verscheidene manieren worden aangepakt.

In de oplossing die hierna volgt, gebruiken we een stapel voor het tijdelijk bewaren van de reeks tekens. De reeks wordt teken na teken gelezen. Elk nieuw teken wordt op de stapel geplaatst. Het lezen eindigt bij de detectie van de aanslag van de RETURN- of ENTER-toets. De tekens worden een voor een van de stapel gehaald en op het beeldscherm weergegeven. De oplossing met een stapel wordt herleid tot de volgende twee stappen :
- lees teken voor teken en zet elk teken op de stapel;
- leeg de stapel teken voor teken en geef elk teken weer.

We lossen het probleem eerst op met Pascal. In de oplossing onderscheiden we drie onderdelen : de definitie van de gegevensstructuren, de definitie van de operaties op deze structuren en tot slot de eigenlijke beschrijving van de oplossing voor het probleem. De definitie van de stapel is :

```
const stackGrootte = 40;

type  StackItem = char;
      Stack     = record
        item : array [1..stackGrootte] of
                         StackItem;
        top  : 0..stackGrootte
      end;
```

Op de structuur definiëren we de volgende bewerkingen :

```
function Leeg : boolean;
{ indien Stack leeg dan TRUE anders FALSE}

procedure Push(x : StackItem);
{ toevoegen van element x op stapel s }

function Pop : StackItem;
{ topelement van stapel s halen }

function Vol : boolean;
{ controleren op overloop van de stapel }
```

Voor de in- en uitvoer gebruiken we de volgende Pascal-procedures :

```
        function Eoln : BOOLEAN;
        { indien ingevoerd teken = ENTER dan TRUE anders FALSE }

        procedure Read(var a : char);
        { invoer van een teken via het toetsenbord }

        procedure Write(a : char);
        { uitvoer van een teken via het beeldscherm }
```

Een deel van het programma voor de oplossing van het probleem is nu :

```
var s     : Stack;
    teken : char;
begin
{ initiëring van de stapel }
s.top := Ø;

{ invoer van de reeks tekens en plaatsen op stapel s }
Read(teken);
while not Vol and not Eoln do
begin
  Push(teken);
  Read(teken)
end;

{ uitvoer van de reeks tekens }
while not Leeg do Write(Pop)
end.
```

In de oplossing gebruiken we de procedure Push en de functies Leeg, Vol en Pop. De kennis van deze procedures en de daarbij horende parameters is echter onvoldoende. In de eigenlijke oplossing van het probleem gebruiken we ook informatie over de wijze waarop de stapel is geïmplementeerd voor de initiëring van de stapel met de opdracht

```
s.top := Ø;
```

De variabele s manipuleren we in de verschillende procedures en in het hoofdprogramma. We definiëren daarom s als een globale variabele. Indien we echter op een later ogenblik de stapel op een andere wijze willen implementeren, bijvoorbeeld met een lijststructuur, dan moeten we ook alle programma-onderdelen aanpassen die deze nieuwe implementatie gebruiken. Voor het programmeren van complexe problemen is het gebruik van globale variabelen

nadelig : de leesbaarheid en de onderhoudbaarheid van de programma's worden sterk verminderd.

De procedures voor de stapel zijn :

```
function Leeg : boolean;

begin
Leeg := s.top = Ø
end {Leeg};

procedure Push(x : StackItem);

begin
s.top := s.top + 1;
s.item[s.top] := x
end {Push};

function Pop : StackItem;

begin
Pop := s.item[s.top];
s.top := s.top - 1
end {Pop};

function Vol : boolean;

begin
Vol := s.top = stackGrootte
end {Vol};
```

De volledige oplossing van het probleem is nu :

```
program Keerom(input,output);
const stackGrootte = 4Ø;
type  StackItem    = char;
      Stack        = record
        item : array [1..stackGrootte] of StackItem;
        top  : Ø..stackGrootte
      end;
var   s             : Stack;
      teken         : char;

function Leeg : boolean;
begin
Leeg := s.top = Ø
end {Leeg};
```

```
procedure Push(x : StackItem);

begin
s.top := s.top + 1;
s.item[s.top] := x
end {Push};

function Pop : StackItem;

begin
Pop := s.item[s.top];
s.top := s.top - 1
end {Pop};

function Vol : boolean;

begin
Vol := s.top = stackGrootte
end {Vol};

begin
{ initiëring van de stapel }
s.top := Ø;

{ invoer van de reeks tekens en plaatsen op stapel s }
Read(teken);
while not Vol and not Eoln do
  begin
  Push(teken);
  Read(teken)
end;

{ uitvoer van de reeks tekens }
while not Leeg do Write(Pop)
end {Keerom}.
```

Een Modula-2 oplossing

We lossen het probleem op dezelfde wijze op als hiervoor. Eerst definiëren we de bewerkingen op een stapel : Pop, Push, Leeg en Vol. Met Modula-2 kunnen we deze gegevensstructuur en de bewerkingen in een afzonderlijke DEFINITION MODULE definiëren.

```
DEFINITION MODULE StackMod;
EXPORT QUALIFIED
  (* proc *) Pop, Push, Leeg, Vol;

PROCEDURE Push(x : CHAR);
PROCEDURE Pop () : CHAR;
PROCEDURE Leeg() : BOOLEAN;
PROCEDURE Vol()  : BOOLEAN;

END StackMod.
```

De EXPORT-lijst van deze module bevat een opsomming van de namen van alle objecten die door de gebruikers van de module StackMod mogen worden gebruikt. De essentie van deze definitiemodule is een abstracte datastructuur. Alle details over de voorstelling van deze structuur worden opgeborgen in de implementatie. De structuur is slechts toegankelijk via de bewerkingen Push, Pop, Leeg en Vol. De bewerkingen Push en Pop manipuleren de structuur, de bewerkingen Leeg en Vol verschaffen informatie over de waarde van de structuur.

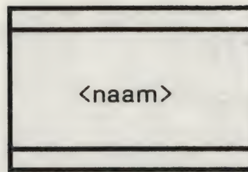
De opsplitsing van de oplossing voor een probleem in afzonderlijke definitie- en implementatiemodulen is een belangrijk onderdeel in het ontwerp van een programmatuursysteem. Elke definitiemodule kan afzonderlijk worden vertaald. Na de vertaling kunnen de objecten in de EXPORT-lijst reeds in andere modulen worden gebruikt. De implementatie van de module staat los van het eigenlijke ontwerp en kan afzonderlijk geschieden.

Voor de in- en uitvoer gebruiken we de abstracte datastructuren die voor de standaardin- en -uitvoer worden gedefinieerd in de module SimpleIO. Hier gebruiken we de bewerkingen ReadChar, WriteChar, WriteLn en EOL.

1.4 Abstractieschema

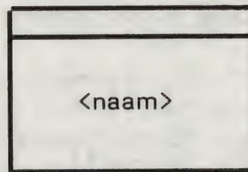
De documentatie van een ontwerp kunnen we samenstellen uit de specificaties van elk van de abstracties en uit een abstractieschema. Een abstractieschema is een grafische voorstelling van de onderlinge samenhang van de verschillende abstracties van een systeem. Een abstractieschema bestaat uit een aantal knooppunten en bogen. Elk knooppunt vertegenwoordigt de abstractie waarvan de naam in het knooppunt wordt vermeld. Een boog geeft aan dat de abstractie aan het begin van de boog moet worden geïmplementeerd met behulp van de abstracties aan het einde van de boog.

Een knooppunt wordt voorgesteld door een rechthoek. Bij een knooppunt voor de voorstelling van een abstract datatype bevat deze rechthoek een dubbel lijnstuk aan de boven- en onderzijde.



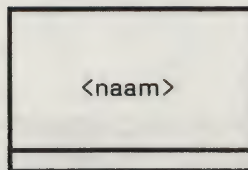
figuur 1.2 Knooppunt voor een abstract datatype

Een abstract datatype wordt met Modula-2 gerealiseerd door middel van een definitie- en implementatiemodule. Soms willen we deze twee modulen in het abstractieschema van elkaar kunnen onderscheiden. Voor de definitiemodule gebruiken we een rechthoek met een dubbel lijnstuk aan de bovenzijde : de definitiemodule beschrijft de interface, de bovenlaag, van de abstractie.



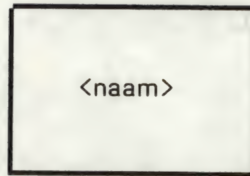
figuur 1.3 Knooppunt voor een definitiemodule

De implementatiemodule stellen we voor als een rechthoek met een dubbel lijnstuk aan de onderzijde : de implementatiemodule bevat de onderliggende details van de abstractie.



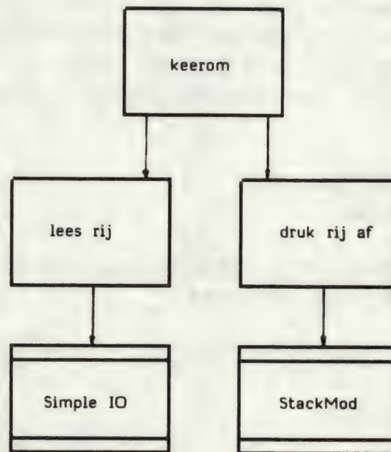
figuur 1.4 Knooppunt voor een implementatiemodule

Een procedure-abstractie stellen we voor als een rechthoek. Een procedure-abstractie realiseren we in Modula-2 met een programmamodule, een procedure of een aantal opdrachten.



figuur 1.5 Knooppunt voor een procedure-abstractie

Voorbeeld : het abstractieschema voor het omkeren van een in te lezen reeks tekens wordt weergegeven in de volgende figuur :



figuur 1.6 Abstractieschema 'keerom'

De uitwerking van de oplossing

We richten onze aandacht weer op de implementatie van de oplossing met Modula-2. Eerst definiëren we de programmamodule Keerom met behulp van de abstracties StackMod en SimpleIO :

```
MODULE Keerom;
FROM StackMod IMPORT Pop, Push, Leeg, Vol;
FROM SimpleIO IMPORT ReadChar, EOL,
                     WriteChar, WriteLn;
VAR  teken : CHAR;

BEGIN
(* invoer van de reeks tekens en plaatsen op de stapel *)
ReadChar(teken);
```

```
WHILE NOT Vol() AND NOT EOL() DO
  Push(teken);
  ReadChar(teken)
END;

(* uitvoer van de reeks tekens *)
WriteLn;
WHILE NOT Leeg() DO
  WriteChar(Pop())
END

END Keerom.
```

We moeten ook nog de gegevensstructuur voor de stapel en de bewerkingen hierop realiseren met een implementatiemodule. We gebruiken dezelfde structuur als in het Pascal-programma. De implementatie voor de stapel kan dan als volgt worden gerealiseerd :

```
IMPLEMENTATION MODULE StackMod;
CONST stackGrootte = 40;
TYPE  Stack        = RECORD
        element : ARRAY [1..stackGrootte] OF CHAR;
        top     : [0..stackGrootte]
      END;
VAR   s            : Stack;

PROCEDURE Push(x : CHAR);

BEGIN
WITH s DO
  top := top + 1;
  element[top] := x
END
END Push;


PROCEDURE Pop() : CHAR;
VAR n : CHAR;

BEGIN
WITH s DO
  n := element[top];
  top := top - 1
END;
RETURN n
END Pop;
```

```
PROCEDURE Leeg() : BOOLEAN;

BEGIN
RETURN s.top = 0
END Leeg;


PROCEDURE Vol() : BOOLEAN;

BEGIN
RETURN s.top = stackGrootte
END Vol;

BEGIN
(* initiëring stapel S *)
s.top := 0
END StackMod.
```

Deze IMPLEMENTATION MODULE bevat niet alleen de implementatie van de bewerkingen op het abstract gegevenstype Stack, maar ook de opdrachten om de datastructuur te initiëren. De gebruiker van een module behoeft zich niet te bekommeren om de toekenning van de juiste beginwaarden aan de variabelen van de implementatie.

1.5 Verborgen typen

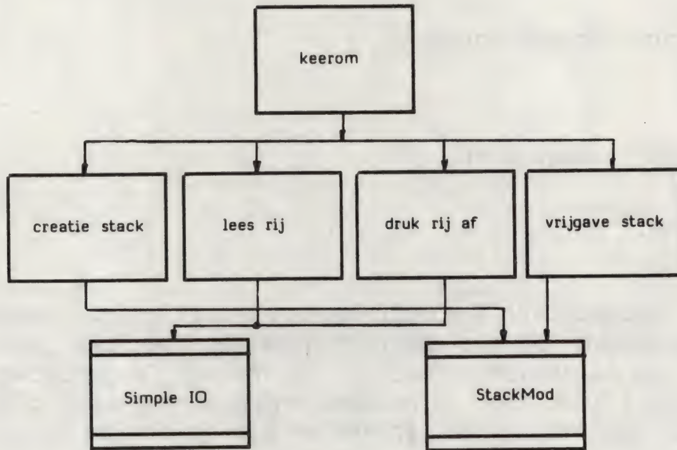
In de voorgaande oplossing zijn alle details van de datastructuur onzichtbaar. Zelfs het type en de typenaam zijn niet toegankelijk. Bij de volgende vorm van de DEFINITION MODULE worden zowel de bewerkingen als het type van de structuur beschikbaar gesteld. Over het algemeen exporteert een Modula-2 module één of meer 'verborgen' typen (E : opaque type, hidden type).

Elke variabele van het 'verborgen' type is dynamisch : dit betekent dat de variabele tijdens de uitvoering van het programma wordt gecreëerd en later wordt vrijgegeven. De oplossing van een probleem waarbij we een dynamische variabele gebruiken, bestaat uit de volgende stappen :
- creëren van de dynamische variabele;
- verwerken met de variabele;
- vrijgeven van de dynamische variabele.

Voor onze probleemstelling kunnen we de volgende oplossing afleiden :

- creëren van de stapel;
- verwerken met de stapel :
    - lees teken voor teken en zet teken op de stapel;
    - leeg de stapel teken voor teken en geef elk teken weer;
- vrijgeven van de stapel.

In deze oplossing gebruiken we nog steeds de abstracte datatypen SimpleIO en Stack. Het abstractieschema voor deze oplossing is :



figuur 1.7 Abstractieschema met verborgen type 'Stack'

De definitie van een abstract datatype Stack

We definiëren in de definitiemodule het type Stack en de bewerkingen CreeerStack, WisStack, Push, Pop, Leeg en Vol. De definitiemodule bevat geen enkele detail over de datastructuur die voor de stapel bij de implementatie wordt gebruikt.

```
DEFINITION MODULE StackMod;
EXPORT QUALIFIED
  (* proc *) Stack, CreeerStack, WisStack, Push, Pop,
             Leeg, Vol;
TYPE Stack; (* verborgen type *)

PROCEDURE CreeerStack(VAR s : Stack);
PROCEDURE WisStack(VAR s : Stack);
PROCEDURE Push(x : CHAR; VAR s : Stack);
PROCEDURE Pop(VAR s : Stack) : CHAR;
PROCEDURE Leeg(s : Stack) : BOOLEAN;
PROCEDURE Vol(s : Stack) : BOOLEAN;
END StackMod.
```

In deze definitiemodule zien we enkele belangrijke verschillen met de voorgaande oplossing :

- de twee operaties CreeerStack en WisStack zijn toegevoegd. De procedure CreeerStack zorgt voor de geheugentoewijzing aan een object van het type Stack en ook voor de initiëring van de waarde van het object. De procedure WisStack zorgt voor het vrijgeven van het geheugen op het ogenblik dat we de stapel niet meer nodig hebben;
- de parameterlijst van elke procedure is aangevuld met een parameter van het type Stack. In een gebruikersmodule kunnen één of meer objecten van het type Stack worden gedeclareerd.

Het gebruik van een Stack

In de gebruikersmodule declareren we eerst een object met

```
VAR a : Stack;
```

Daarna wordt de stapel gecreëerd met de operatie

```
CreeerStack(a);
```

Vanaf nu kunnen we de andere operaties Push, Pop, Leeg en Vol op de stapel uitvoeren. Tot slot geven we de stapel vrij met de opdracht

```
WisStack(a);
```

De oplossing van het probleem is nu :

```
MODULE Keerom;
FROM StackMod IMPORT Stack, CreeerStack, WisStack, Pop, Push,
                     Leeg, Vol;
FROM SimpleIO IMPORT ReadChar, WriteChar, WriteLn, EOL;

VAR  teken : CHAR;
     a     : Stack;

BEGIN
CreeerStack(a);

(* invoer van de reeks tekens en plaatsen op de stapel *)
ReadChar(teken);
WHILE NOT Vol(a) AND NOT EOL() DO
  Push(teken,a);
  ReadChar(teken);
END;
```

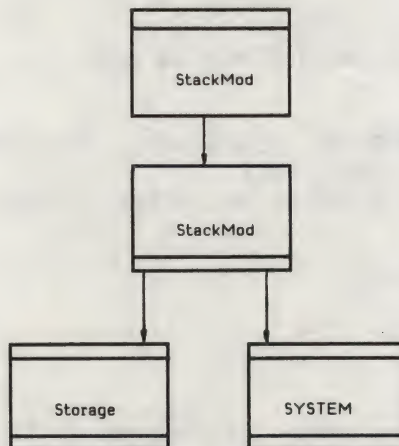
```
(* uitvoer van de reeks tekens *)
WriteLn;
WHILE NOT Leeg(a) DO
  WriteChar(Pop(a))
END;

WisStack(a)
END Keerom.
```

Het belangrijkste verschil met de vorige oplossing is dat nu de variabelen van een verborgen type in de gebruikersmodule zelf worden gedefinieerd, terwijl in de vorige oplossing de variabele zelf onzichtbaar is.

Implementatie van de Stack

Een verborgen type wordt steeds als een dynamische variabele geïmplementeerd. We beheren het geheugen met de bewerkingen ALLOCATE en DEALLOCATE van het abstracte type 'Storage'. Met de functie 'TSIZE' uit de module 'SYSTEM' bepalen we de hoeveelheid geheugenruimte die aan een Stack moet worden toegewezen. Voor het abstracte datatype Stack geldt het volgende abstractieschema :



figuur 1.8 Implementatie Stack

```
IMPLEMENTATION MODULE StackMod;
FROM Storage IMPORT ALLOCATE, DEALLOCATE;
FROM SYSTEM  IMPORT TSIZE;

CONST stackGrootte = 40;
TYPE  StackType    = RECORD
        element : ARRAY [1..stackGrootte] OF CHAR;
        top     : [0..stackGrootte]
      END;
      Stack        = POINTER TO StackType;


PROCEDURE CreeerStack(VAR s : Stack);

BEGIN
ALLOCATE(s, TSIZE(Stack));
s^.top := 0
END CreeerStack;

PROCEDURE WisStack(VAR s : Stack);

BEGIN
DEALLOCATE(s, TSIZE(Stack))
END WisStack;


PROCEDURE Push(    x : CHAR;
               VAR s : Stack);

BEGIN
WITH s^ DO
  top := top + 1;
  element[top] := x
END
END Push;

PROCEDURE Pop(VAR s : Stack) : CHAR;
VAR n : CHAR;

BEGIN
WITH s^ DO
  n := element[top];
  top := top - 1
END;
RETURN n
END Pop;
```

```
PROCEDURE Leeg(s : Stack) : BOOLEAN;

BEGIN
RETURN s^.top = 0
END Leeg;


PROCEDURE Vol(s : Stack) : BOOLEAN;

BEGIN
RETURN s^.top = stackGrootte
END Vol;


BEGIN
END StackMod.
```

1.6 Een Modula-2 ontwikkelingsomgeving

Ervaren mensen die op verschillende gebieden (geneeskunde, wetgeving, techniek) werkzaam zijn, gebruiken hulpmiddelen om hun produktiviteit te verhogen. De software-ontwikkelaar vormt hierop geen uitzondering. Verscheidene hulpmiddelen zijn ontwikkeld ter ondersteuning van de activiteiten die tijdens de ontwikkeling worden uitgevoerd. Enkele belangrijke programmeerhulpmiddelen zijn besturingssystemen, compilers, vertolkers, editors, koppelprogramma's, 'pretty printers', symbolische debuggers en programma's voor het opstellen van 'cross reference'-tabellen.

De modulaire ondersteuning door een Modula-2 systeem vereist een bijzondere ontwikkelingsomgeving. In het bijzonder vormen de compiler, het koppelprogramma, de modulebibliotheek, de 'loader' en het 'runtime'-systeem een geïntegreerd geheel. Indien we een module aan de bibliotheek willen toevoegen, is het voldoende deze module te compileren. De compiler gebruikt automatisch de definitiemodulen uit de bibliotheek, indien hieraan wordt gerefereerd. Het koppelprogramma koppelt automatisch alle implementatiemodulen tot een uitvoerbaar geheel. Zowel de compiler als het koppelprogramma genereren hulpbestanden voor de symbolische debugger zodat we een geheugendump van een onderbroken programma kunnen analyseren.

1.6.1. De compiler

Compileereenheden

Modula-2 definieert drie typen compileereenheden : programmamodule, definitiemodule en implementatiemodule. Een programmamodule stelt een uitvoerbaar programma samen. Een programmamodule dient als hoofdprogramma of als een subprogramma dat door een andere module wordt aangeroepen. De algemene vorm is :

```
MODULE P;
IMPORT ... ;

CONST ...;
TYPE  ...;
VAR   ...;

  PROCEDURE P1(...);
  ...
  BEGIN
    ...
  END P1;

BEGIN
(* verwerking P *)
...
END P.
```

Een bibliotheekmodule bestaat uit twee afzonderlijk gecompileerde modulen : de DEFINITION MODULE en de corresponderende IMPLEMENTATION MODULE. Een compileereenheid die andere modulen invoert, heeft tijdens de compilatie uitsluitend toegang nodig tot de definitiemodulen. De implementatiemodulen kunnen aldus opnieuw worden vertaald zonder dat de rest van het systeem wordt beïnvloed. De algemene vorm voor deze modulen is :

```
DEFINITION MODULE M;
EXPORT QUALIFIED M1, M2, ...;

(* definitie objecten van module M *)

END M.
```

```
IMPLEMENTATION MODULE M;
FROM N IMPORT N1, N2, ...;

(* implementatie objecten van module M *)

BEGIN
(* initiëring datastructuren van module M *)

END M.
```

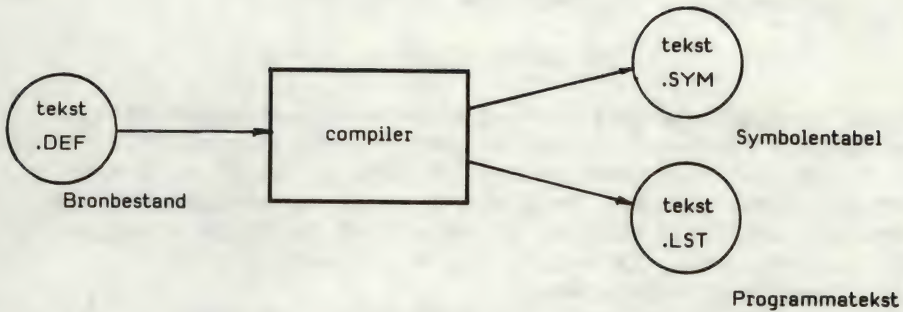
In- en uitvoerbestanden voor de compiler

De verschillende software-hulpmiddelen in een Modula-2 omgeving hebben een of meer invoerbestanden nodig en maken een of meer uitvoerbestanden aan. Elk van deze bestanden heeft een bestandsnaam. De keuze van de bestandsnamen is afhankelijk van het onderliggende besturingssysteem en van de Modula-2 implementatie. Over het algemeen bestaat de bestandsnaam uit twee delen : de eigenlijke naam van het bestand en de uitbreiding. Met de uitbreiding kunnen we het type van een bestand aangeven. De uitbreidingen voor de verschillende typen van bestanden worden automatisch gekozen. De systeemgekozen waarden voor de uitbreiding die hierna worden vermeld zijn typisch voor Modula-2 implementaties, bijvoorbeeld de Lilith-implementatie.

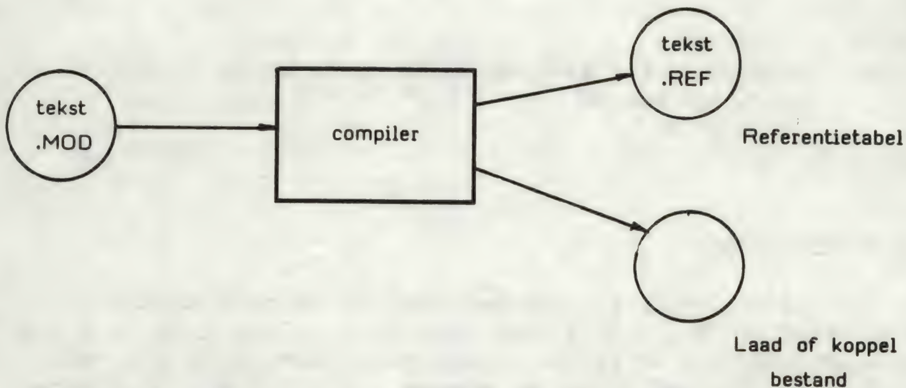
Tijdens de verwerking van een compileereenheid heeft de compiler een bronbestand nodig. Het bronbestand bevat de tekst van de compileereenheid. De namen van de bronbestanden worden meestal getypeerd door de uitbreiding 'MOD' voor de programma- en de implementatiemodulen en 'DEF' voor de definitiemodulen. De systeemgekozen waarde is 'MOD'.

De uitvoer van de compiler kan de volgende bestanden bevatten :

- programmatekst : de tekst van de compileereenheid eventueel aangevuld met foutboodschappen en aanvullende informatie; de systeemgekozen uitbreiding is 'LST';
- symbolentabel : een bestand met informatie voor de opbouw van een symbolentabel. Dit bestand wordt aangemaakt tijdens de compilatie van een definitiemodule. De systeemgekozen uitbreiding is 'SYM';
- referentiebestand : een bestand met informatie voor de symbolische debugger. Het bestand wordt aangemaakt tijdens de compilatie van een progamma of implementatiemodule. De systeemgekozen uitbreiding is 'REF';
- doelbestand : het bestand met de gegenereerde code voor het laad- of het koppelprogramma.



figuur 1.9 Aanmaak van een bibliotheekmodule



figuur 1.10 Aanmaak van een implementatie

Compileropties

Compileropties worden opgenomen in het te compileren bronprogramma. Deze opties geven bijzondere opdrachten aan het vertaalprogramma tijdens de compilatie. Zodra het vertaalprogramma de optie tegenkomt in het bronprogramma worden deze opdrachten uitgevoerd. De syntaxis voor een optie is :

```
(*$<letter><instelling>[<andere tekst>]*)
```

Spaties zijn niet toegelaten in het deel

'(*$<letter><instelling>'.

De opties kunnen worden aan- of afgezet of teruggezet naar de voorlaatste toestand. Hiervoor gebruiken we respectievelijk het '+'-teken, het '-'-teken of het '='-teken. De opties hebben een systeemgekozen waarde. Deze waarde wordt automatisch door de compiler gebruikt indien we de optie niet expliciet vermelden in het bronprogramma. De belangrijkste opties zijn :
- S : test op stapeloverloop;
- R : test op deelinterval en rekenkundige overloop;
- T : test op index van tabellen en van CASE-opdrachten.

Voorbeeld :

```
MODULE X;
(*$T+   testcode wordt aangemaakt *)
...
...
(*$T-   geen testcode *)
...
...
(*$T=   testcode wordt aangemaakt; de optie wordt teruggezet naar
        de vorige waarde *)

END X.
```

De modulesleutel

Bij elke compilatie van een compileereenheid wordt een modulesleutel (E : module key) aangemaakt. Deze sleutel is uniek en wordt gebruikt om verschillende gecompileerde versies van dezelfde module van elkaar te kunnen onderscheiden. De sleutel wordt geregistreerd in het bestand met de symbolentabel en in het doelbestand. Indien de sleutels van een definitie- en van een implementatiemodule niet met elkaar overeenkomen, wordt een foutboodschap gegeven tijdens de compilatie of tijdens het koppelen of laden van het programma.

1.6.2. Versiecontrole

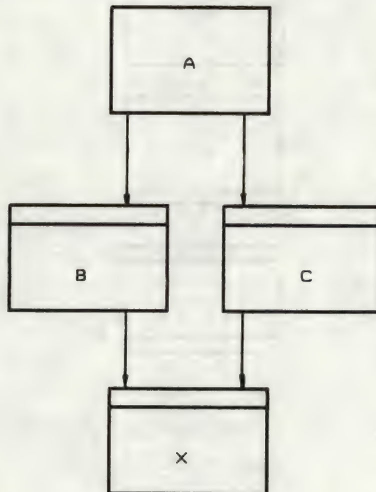
Alle modulen in een programma moeten worden gecompileerd met consistente versies van definitiemodulen. Wanneer we een definitiemodule wijzigen moeten we ook alle programma- en implementatiemodulen, die gebruiker zijn van deze definitiemodule, opnieuw compileren voordat we een uitvoerbaar programma aanmaken. Elke wijziging van een definitiemodule impliceert een nieuwe compilatie van die module en de aanmaak van een nieuw bestand met de symbolentabel. Bij elke compilatie wordt een nieuwe modulesleutel in dit bestand geschreven. Deze sleutel is uniek zelfs indien we de definitiemodule niet hebben gewijzigd.

Tijdens de compilatie, het koppelen of het laden van een programma worden alle modulesleutels voor een gegeven definitiemodule gecontroleerd op gelijkheid. Hierdoor krijgen we de zekerheid dat verschillende modulen, die een definitiemodule gemeenschappelijk gebruiken, zijn vertaald met dezelfde versie van die definitiemodule.

Versiecontrole van modulen is in een Modula-2 ontwikkelingsomgeving analoog met de type-controle tijdens de compilatie. Voor Modula-2 zijn beide controles even belangrijk.

Versiefouten tijdens de compilatie

Veronderstel we hebben de modulen A, B, C en X. Module A is een gebruiker van B en C. Modulen B en C zijn beide gebruiker van module X. We stellen dit voor met het volgende schema :



figuur 1.11 Versiecontrole tijdens het compileren

We compileren deze modulen in de volgende volgorde :

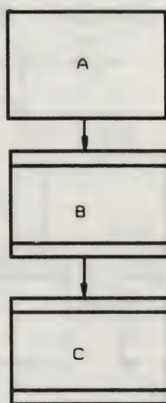
```
X.DEF  =>  X.SYM
B.DEF  =>  B.SYM
X.DEF  =>  X.SYM
C.DEF  =>  C.SYM
A.MOD  =>  ...
```

Tijdens de verwerking van module A ontdekt de compiler een versieconflict : module B gebruikt immers een andere versie van X dan module C. Voordat we module A verder kunnen verwerken dienen we module B opnieuw te compileren.

In het algemeen komen versiefouten tijdens de compilatie slechts voor wanneer een module niet rechtstreeks langs twee of meer verschillende paden gebruiker is van een gemeenschappelijke module en wanneer de versie van deze gemeenschappelijke module niet voor alle paden identiek is.

Versiefouten tijdens het koppelen of laden van een programma

Veronderstel we hebben de bibliotheekmodulen B en C en een programmamodule A. We stellen de onderlinge afhankelijkheid voor met de volgende figuur :



figuur 1.12 Versiecontrole tijdens het koppelen of laden

We voeren de volgende compilaties uit :

```
C.DEF   =>  C.SYM
B.DEF   =>  B.SYM
B.MOD   =>  B.LNK
A.MOD   =>  A.LNK
C.DEF   =>  C.SYM  (bron van inconsistentie)
C.MOD   =>  C.LNK
```

In een volgende fase koppelen we de modulen C.LNK, B.LNK en A.LNK tot een uitvoerbaar geheel. Tijdens de verwerking ontstaat een versieconflict tussen de module A.LNK (gebruikt C.SYM versie 1) en C.LNK (gebruikt C.SYM versie 2).

Een versieconflict tijdens het koppelen of laden van een programma is slechts mogelijk indien in modulen verschillende versies van een gemeenschappelijke definitiemodule worden gebruikt.

Deze problemen kunnen we voorkomen indien we de volgende regels naleven :
- een definitiemodule moet steeds voor haar gebruikersmodulen worden vertaald;
- een implementatiemodule kan onafhankelijk van alle andere modulen van het systeem worden vertaald;
- als we een definitiemodule opnieuw compileren, moeten we ook alle gebruikers van de module opnieuw compileren.

1.6.3. De symbolische debugger

De symbolische debugger is een hulpmiddel om een slecht werkend programma te onderzoeken : we kunnen nagaan waar en waarom een programma verkeerd loopt. Tijdens het onderzoek gebruiken we de objecten zoals die in de tekst 'symbolisch' zijn gedefinieerd. De belangrijkste twee typen zijn de 'post mortem debugger' en de interactieve of dynamische debugger. Beide typen verschaffen gelijksoortige informatie over het programma.

Met een interactieve debugger onderzoeken we een programma tijdens de werking. In het programma plaatsen we vooraf breekpunten zodat het programma daar automatisch stopt. Op de plaats van het breekpunt kunnen we de waarden van de programma-objecten opvragen en eventueel wijzigen met daartoe bestemde opdrachten. Op elk ogenblik kan het programma opnieuw worden gestart.

Als een programma abnormaal eindigt registreert het Modula-2 systeem automatisch de inhoud van het geheugen in een 'post mortem dump'-bestand. We kunnen dit bestand analyseren met behulp van de symbolische 'post mortem debugger'. Het systeem creëert een geheugendump in de volgende gevallen : bij een uitvoeringsfout tijdens de werking van het programma, bij een aanroep van de

procedure HALT of bij het met de hand onderbreken van het programma. Voor de analyse van de geheugendump gebruikt de debugger ook nog de volgende bestanden : een referentiebestand dat wordt aangemaakt tijdens de compilatie van de implementatie- of van de programmamodulen, eventueel de bronbestanden van de bibliotheek en van het hoofdprogramma.

De debugger kan verschillende typen informatie verschaffen over het programma. Elk type informatie wordt weergegeven in een venster. De belangrijkste informatievensters zijn :

- procedurevenster :

Elk Modula-2 programma bevat ten minste één proces, het hoofdprogramma zelf. Het procedurevenster toont de keten van procedure-aanroepen van het proces dat in werking is op het ogenblik dat het programma wordt onderbroken. We kunnen in deze keten een procedure selecteren en aldus de tekst of de gegevens die bij de procedure behoren in detail onderzoeken.

- modulevenster :

Het modulevenster toont een lijst met modulen waaruit het programma is opgebouwd. We kunnen een module selecteren en aldus de tekst of de gegevens die bij de module behoren nader onderzoeken.

- datavenster :

Het datavenster toont de waarde van de variabelen die bij een procedure of een module behoren. De procedure of de module kiezen we vooraf met het procedure- of modulevenster. De debugger bevat operaties voor de toegang tot de elementen van een gestructureerd datatype.

- tekstvenster :

Het tekstvenster toont de tekst van een procedure of een module. De procedure of de module kiezen we vooraf met het procedure- of modulevenster. De debugger bevat operaties voor het doorlopen van de tekst.

- ruwe venster :

Het ruwe venster toont de geheugeninhoud die is geregistreerd op het ogenblik dat het programma is onderbroken. De debugger bevat operaties voor de vorm van de weergave (hexadecimaal, decimaal of tekst) of voor de toegang tot een bepaald adres.

Tot slot

Modula-2 is niet zozeer een programmeertaal voor algemeen gebruik maar eerder een taal voor de ontwikkeling van systeemprogrammatuur. Met Modula-2 kunnen we dezelfde verzameling programma's formuleren als met Pascal, C of andere systeemtalen, maar met de ondersteuning van het programmeren met modulen en abstracte datatypen. Modula-2 is ook zeer geschikt voor het programmeeronderwijs aan toekomstige software-ontwikkelaars : de stap van Pascal naar Modula-2 is gering en vereist weinig inspanning. Bovendien zijn Modula-2 implementaties beschikbaar voor besturingssystemen die op de populairste microcomputers draaien.

Literatuur

Ford G. A., Wiener R., 'Modula-2, a Software Development Approach',
Wiley, New York 1985.

Wiener R., Sincovec R., 'Software Engineering with Modula-2 and ADA', Wiley, New York 1984.

Wirth N., 'Programming in Modula-2', derde verbeterde editie
Springer Verlag, Berlijn 1985.

Oefeningen

1. Bestudeer de Modula-2 ontwikkelingsomgeving die op je computersysteem draait :

- welke regels worden opgelegd voor de keuze van bestandsnamen voor een programma- , een definitie- en een implementatiemodule ?
- met welke opdrachten kan je een programma- , een definitie- of een implementatiemodule compileren ? Over welke bestanden moet je beschikken voor deze compilaties ?
- hoe kan je de programmatekst met foutboodschappen, die door de compiler wordt aangemaakt, raadplegen ?
- met welke opdrachten worden een programmamodule en de implementatiemodulen samengeknoopt tot een uitvoerbaar geheel ? Over welke bestanden moet je beschikken voor deze opdrachten ?
- over welke hulpmiddelen kan je beschikken om een programma uit te vlooien (bijvoorbeeld : cross reference, symbolische debugger). Hoe moet je deze hulpmiddelen gebruiken ?

2. Bestudeer de bibliotheekmodule voor de vereenvoudigde standaardin- en -uitvoer. Zoek de opdrachten op voor het lezen en schrijven van een geheel getal met type CARDINAL.

3. Maak een tekstbestand aan voor de volgende programmamodule :

```
MODULE Som;
FROM SimpleIO IMPORT WriteCard, WriteLn;

TYPE   Getal = [Ø..1Ø];
VAR    i     : Getal;
       som   : Getal;

BEGIN
som := Ø;
i := 1;
WHILE i <= 1Ø DO
  som := som + i;
  INC(i)
END;
WriteCard(som, 5);
WriteLn
END Som.
```

- Compileer en koppel dit programma tot een uitvoerbaar programma.
- Laat dit programma draaien op je computersysteem.
- Analyseer de foutboodschappen tijdens de uitvoering van het programma.
- Gebruik zo mogelijk een symbolische debugger om de fout in de tekst te lokaliseren. Onderzoek ook de waarden van de variabelen op het ogenblik dat de uitvoering is gestopt.

2 Modula-2 woordenschat

Het ontwerpvoorbeeld uit het vorige hoofdstuk illustreert duidelijk dat we met Modula-2 op een eenvoudige wijze abstracte datatypen kunnen definiëren en toepassen voor het oplossen van een probleem. Dit voorbeeld is vooral bedoeld ter kennismaking; we hebben ons nog niet bekommerd om de syntaxis en de semantiek van de taal. In dit hoofdstuk besteden we juist hieraan de nodige aandacht. De eerste paragraaf behandelt een beschrijvende taal waarmee we later de syntaxis van Modula-2 definiëren. In de volgende paragraaf wordt de woordenschat beschreven. Hier en daar worden ook enige stijlvoorschriften vermeld.

2.1 Syntaxisnotatie

Een programma is samengesteld uit een aantal onderdelen zoals opdrachten, declaraties, uitdrukkingen. Deze onderdelen noemen we de constructies van de taal. Deze constructies moeten heel precies voldoen aan een aantal voorschriften inzake de notatie. De syntaxis van een taal is de verzameling van deze voorschriften. We kunnen de syntaxis van een taal beschrijven met een beschrijvende taal (meta-notatie), met syntaxisdiagrammen of met verklarende tekst.

De syntaxis van Modula-2 wordt beschreven met de meta-notatie Extended Backus-Naur Form (EBNF). In EBNF krijgt elke constructie een naam. De keuze van deze namen is vrij en sluit zo goed mogelijk aan bij de constructie die we willen benoemen. Als een naam slechts uit één woord bestaat, schrijven we hem steeds met kleine letters :

identifier, letter, cijfer, operator

Er zijn ook namen die bestaan uit samengestelde woorden. In dit geval is de eerste letter van elk woord in de samenstelling een hoofdletter :

ScheidingsTeken, SchaalFactor, KleineLetter

De meeste constructies worden gedefinieerd met behulp van andere constructies. Sommige constructies zijn atomair : we kunnen deze niet beschrijven met behulp van andere constructies. Deze atomaire constructies worden eindtermen genoemd. De waarde van een eindterm wordt voorafgegaan en gevolgd door een aanhalingsteken " of '. De eindtermen voor een cijfer zijn :

'Ø' '1' '2' '3' '4' '5' '6' '7' '8' '9'

Een produktieregel is de formulering van een constructie. Een produktieregel bestaat uit de naam van de constructie die we willen definiëren, het teken '=' met de betekenis 'wordt gedefinieerd door', en een opeenvolging van constructies. Als een constructie a wordt samengesteld uit de opeenvolging van de constructies b en c dan zijn b en c syntactische factoren. We noteren dit als

a = b c

Zo betekent

identifier = letter cijfer

dat elke identifier bestaat uit de samenvoeging van een letter en een cijfer. Als een constructie wordt gevormd uit de constructie b of de constructie c dan zijn b en c syntactische termen. Voor de aanduiding van de alternatieve keuze van een syntactische term gebruiken we het symbool '|'.

a = b | c

Voor de constructie 'cijfer' geldt de produktieregel

```
cijfer = 'Ø' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
```

Voor de letters onderscheiden we de kleine letters en de hoofdletters :

```
KleineLetter = 'a' | 'b' | 'c' | ... | 'z'
HoofdLetter  = 'A' | 'B' | 'C' | ... | 'Z'
letter       = KleineLetter | HoofdLetter
```

Met de tekens '|' en '=' kunnen we dus de constructies zeer precies beschrijven. Deze tekens behoren tot de metasymbolen van de beschrijvende taal. Met de tekens '...' willen we aangeven dat ook alle letters tussen 'c' en 'z' eindtermen zijn. De tekenreeks '...' behoort niet tot EBNF. Andere metasymbolen die worden gebruikt zijn :

[ ] de rechte haken; ze omsluiten een keuze uit een aantal constructies;

{ } de accolades; ze omsluiten constructies die nul of meer keer mogen worden herhaald;

( ) de haakjes; ze groeperen constructies.

We beschikken nu over voldoende gereedschap om alle Modula-2 produktieregels te formuleren. We illustreren dit met de definitie van een identifier :

identifier = letter { letter | cijfer }

Ter aanvulling van deze EBNF-notatie definiëren we de produktieregels ook met syntaxisdiagrammen. Een syntaxisdiagram is een grafische voorstelling voor een constructie. Elke eindterm wordt voorgesteld door een rechthoek met afgeronde hoeken of met een cirkel. Elke niet-eindterm wordt voorgesteld door een rechthoek. De grafische elementen worden verbonden door pijlen. Deze pijlen geven het pad aan dat we mogen volgen bij de samenstelling van een constructie.

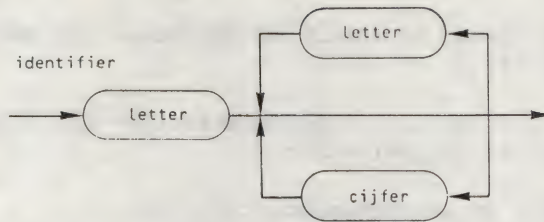
2.2 Woordenschat

De Modula-2 woordenschat is samengesteld uit identifiers, gereserveerde woorden en symbolen, en commentaar.

Identifier

Een identifier is een symbool om een Modula-2 object aan te duiden : een constante, een variabele, een procedure, een type. Een identifier wordt voorgesteld door een string. Deze string begint steeds met een letter en wordt eventueel gevolgd door een willekeurig aantal letters of cijfers.

identifier = letter { letter | cijfer }



Syntaxisdiagram identifier

Voorbeelden van identifiers zijn :
N
aanwezig
AantalAanwezig
som
A12

In de identifier wordt een onderscheid gemaakt tussen de hoofdletters en de kleine letters. De identifiers

Som

en

som

zijn voor de compiler verschillend. Bij het programmeren moeten we met deze eigenschap rekening houden, zodat we fouten kunnen voorkomen.
De lengte van de identifiers is in principe onbeperkt : alle tekens van de identifier zijn significant. De vrijheid in de keuze van identifiers is daarom zeer groot. Deze vrijheid kan de leesbaarheid van de programma's sterk verbeteren, indien we de identifiers op verantwoorde wijze kiezen. De volgende stijlregels worden aanbevolen :

- ingewikkelde namen, samengesteld uit meer woorden, dienen kleine letters en hoofdletters te bevatten. De hoofdletters worden gebruikt om de woordengrenzen te kunnen onderscheiden :
BalansRekening
AantalSectoren
som

- namen voor typen, procedures en modulen beginnen met een hoofdletter :
BerekenBalans
StapelMod
File
Push

- de namen voor parameters, variabelen, velden van een record en constanten beginnen met een kleine letter :
verschuldigdBedrag
i
x
element

- een procedurenaam bevat steeds een werkwoordsvorm :
Pop
Push
CreeerStapel
BerekenSaldo

- de naam van de programmamodule kan zowel hoofdletters als kleine letters bevatten. Deze naam kan eventueel afhankelijk zijn van het onderliggende besturingssysteem.

Standaardidentifiers

Sommige identifiers, de standaardidentifiers, kunnen we zonder voorafgaande declaraties in een programma gebruiken. De standaardidentifiers zijn :

ABS	BITSET	BOOLEAN	CAP	CARDINAL
CHAR	CHR	DEC	DISPOSE	EXCL
FALSE	FLOAT	HALT	HIGH	INC
INCL	INTEGER	LONGCARD	LONGINT	LONGREAL
MAX	MIN	NEW	NIL	ODD
ORD	PROC	REAL	SIZE	TRUE
TRUNC	VAL			

Deze identifiers worden in de loop van de tekst verklaard.

Getallen

Getallen worden op twee manieren in een programma geschreven : de voorstellingswijze voor een geheel getal (integer) en de voorstellingswijze voor een reële benadering, meestal geen geheel getal, van een hoeveelheid (real).

Gehele getallen

Een geheel getal kunnen we voorstellen als een octale, decimale, hexadecimale numerieke constante of als een numeriek tekensymbool.

- Een octaal getal begint met een octaal cijfer van 0 tot 7, wordt eventueel gevolgd door een willekeurig aantal octale cijfers, en eindigt met de letter 'B'.

```
OctCijfer = '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7'
OctGetal  = OctCijfer {OctCijfer} 'B'
```

Voorbeelden :

0B   37B   1567B   1765B

- Een decimaal getal begint met een decimaal cijfer van 0 tot 9 en wordt eventueel gevolgd door een willekeurig aantal decimale cijfers.

```
DecimaalGetal = cijfer { cijfer }
```

Voorbeelden :

0   1678   4450100

- Een hexadecimaal getal begint met een decimaal cijfer, wordt eventueel gevolgd door een willekeurig aantal hexadecimale cijfers, en eindigt met de letter 'H'.

```
HexCijfer = cijfer | 'A' | 'B' | 'C' | 'D' | 'E' | 'F'
HexGetal  = cijfer { HexCijfer } 'H'
```

Voorbeelden :

0H   23EFH   0FFFFH

- Een numerieke tekenconstante begint met een octaal cijfer, wordt eventueel gevolgd door een aantal octale cijfers, en eindigt met de letter 'C'.

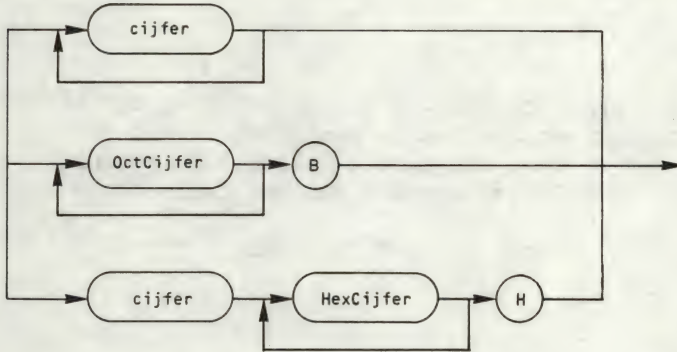
```
TekenConstante = OctCijfer { OctCijfer } 'C'
```

Voorbeelden :

0C   36C   127C

We kunnen de syntaxis voor een integer onderbrengen in de volgende regel :

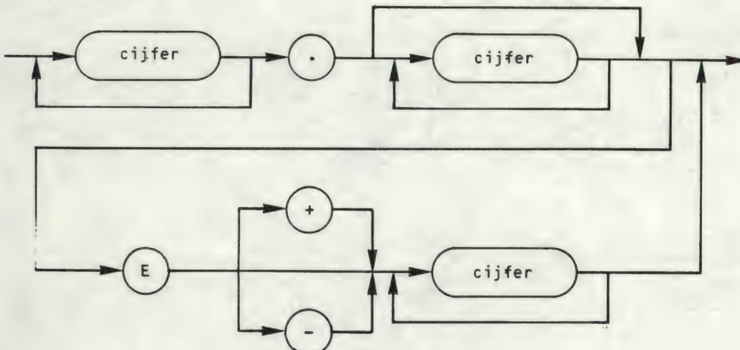
```
IntegerConstante = cijfer { cijfer }
                 | OctCijfer { OctCijfer } ( 'B' | 'C' )
                 | cijfer { HexCijfer } 'H'
```



Reële getallen

Een reëel getal begint met een decimaal cijfer en wordt eventueel gevolgd door een willekeurig aantal cijfers. Het getal moet precies één decimale punt bevatten. Het getal mag eindigen met een schaalfactor. Deze schaalfactor bestaat uit de letter 'E', gevolgd door naar keuze een '+'- of '-'-teken en een of meer decimale tekens. De waarde van de schaalfactor geeft de macht van tien aan waarmee het getal moet worden vermenigvuldigd. Zo betekent 'E+12' : 'vermenigvuldigd met 10 tot de macht 12'.

```
SchaalFactor    = 'E' [ '+' | '-' ] cijfer { cijfer }
ReëleConstante = cijfer { cijfer } '.' { cijfer } [ SchaalFactor ]
```



Voorbeelden :

1.Ø 1. 1.23E+5 1.4E1Ø 6.Ø23E-23

De algemene regel voor het vormen van een getal wordt gegeven door :

getal = integer | real

Strings

Een string is een opeenvolging van een willekeurig aantal tekens. Een string begint met een van de aanhalingstekens "'" of '"'. Daarna volgen nul of meer afdrukbare tekens. De string eindigt met hetzelfde aanhalingsteken als aan het begin.

string = '"' { teken } '"' | "'" { teken } "'"

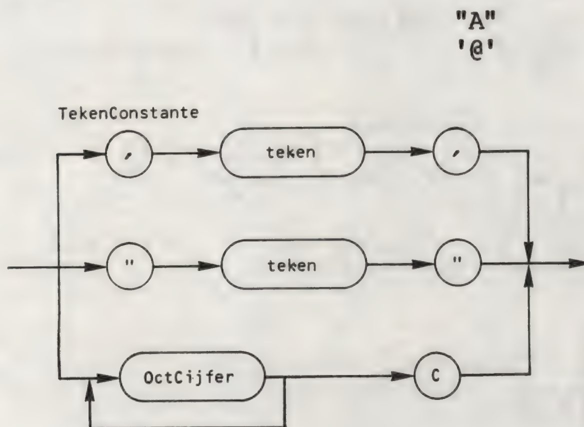
'de ggd van '
"dit is het einde"
""

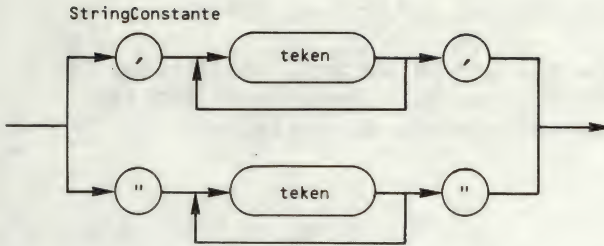
In de string kunnen aanhalingstekens voorkomen. Deze tekens moeten verschillen van de aanhalingstekens die de string begrenzen.

'Het antwoord is "hello"'
"Antwoord 'J' of 'N'"

De lengte van de string is gelijk aan het aantal tekens van de string, de begrenzingstekens niet inbegrepen. Een string met lengte 1 is een tekenconstante.

"A"
'@'





Begrenzingstekens

Begrenzingstekens zijn symbolen die twee taalelementen van elkaar scheiden. Voor en na de begrenzingstekens hoeven we geen spaties op te nemen. Het gebruik van spaties voor en na de begrenzingstekens verbetert echter de leesbaarheid van het programma. Sommige begrenzingssymbolen worden samengesteld uit twee tekens. Deze twee tekens mogen niet van elkaar worden gescheiden door spaties of een teken voor een nieuwe regel.

De betekenis van de begrenzingstekens is :

+	optelling, vereniging van verzamelingen;
-	aftrekking, verschil van verzamelingen, monadische operator;
*	vermenigvuldiging, doorsnede van verzamelingen;
/	deling, symmetrisch verschil van verzamelingen;
:=	toekenning;
&	logische AND;
~	logische NOT;
=	gelijk aan;
# <>	niet gelijk aan;
<	kleiner dan;
>	groter dan;
<=	kleiner dan of gelijk aan;
>=	groter dan of gelijk aan;
( )	haakjes voor uitdrukkingen;
[ ]	haakjes voor indices, deelinterval;
{ }	haakjes voor verzamelingen;
(* *)	haakjes voor commentaar;
^	referentie-operator;
,	interpunctie; scheiding van de elementen van een lijst;
.	interpunctie; einde van de module; scheiding van de onderdelen van gekwalificeerde namen;

; interpunctie; scheiding van opdrachten; afsluiting van declaraties;
: interpunctie; definitie van parameters, variabelen;
| interpunctie; scheiding van de alternatieven van een CASE-opdracht of van een record met varianten.

Gereserveerde woorden

De gereserveerde woorden worden opgesomd in de volgende lijst. Zij worden steeds met hoofdletters geschreven.

AND	ARRAY	BEGIN	BY	CASE
CONST	DEFINITION	DIV	DO	ELSE
ELSIF	END	EXIT	EXPORT	FOR
FROM	IF	IMPLEMENTATION	IMPORT	IN
LOOP	MOD	MODULE	NOT	OF
OR	POINTER	PROCEDURE	QUALIFIED	RECORD
REPEAT	RETURN	SET	THEN	TO
TYPE	UNTIL	VAR	WHILE	WITH

De betekenis van de gereserveerde woorden wordt in de loop van de tekst verklaard.

Commentaar

Een Modula-2 programma wordt beschouwd als een opeenvolging van identifiers, getallen, strings, begrenzingstekens en gereserveerde woorden zoals die in de voorafgaande paragrafen zijn beschreven. In het algemeen mag een willekeurig aantal spaties of tekens voor de aanduiding 'einde regel' tussen de verschillende taalelementen voorkomen. Met een weloverwogen lay-out en keuze van de identifiers is een Modula-2 programma in hoofdzaak zelfverklarend. Hier en daar is het echter nog steeds noodzakelijk aanvullende uitleg en commentaar aan de programmatekst toe te voegen.

Het begin van het commentaar wordt aangegeven met het symbool '(*', het einde met het symbool '*)'. De tekst tussen deze beide symbolen wordt door de compiler niet geïnterpreteerd. Het commentaar mag over meer regels worden verdeeld.

```
(* willekeurige tekst *)
```

Het commentaar is toegelaten op elke plaats waar in de programmatekst spaties of tekens voor de aanduiding 'nieuwe regel' mogen voorkomen.

Commentaarteksten mogen worden genest : dit betekent dat we in commentaar een nieuwe commentaartekst mogen schrijven. Voor de

compiler dient het aantal openingssymbolen voor commentaar gelijk te zijn aan het aantal afsluitingssymbolen.

```
(* begin commentaar
...
(* begin geneste commentaar    *)

...
einde commentaar *)
```

Het nesten van commentaar biedt enkele belangrijke voordelen bij het ontwikkelen van programma's :

- bij een verkeerde afsluiting van commentaar wordt een gedeelte van de programmatekst ongewild door andere compilers (ondermeer Pascal) niet geïnterpreteerd. Bijvoorbeeld :

```
...
i := j - k;    (* onthoud verschil
j := k;        (* herstel j *)
...
```

Voor deze compilers begint het commentaar op de eerste regel en eindigt op de tweede regel. De opdracht 'j := k;' wordt stilzwijgend tot het commentaar gerekend en verdwijnt dus uit het programma. Deze fout is meestal zeer moeilijk te ontdekken. Een Modula-2 compiler ontdekt bij dit type fouten dat er geen evenwicht tussen de openende en sluitende commentaarsymbolen is. Tijdens de compilatie wordt een syntaxis-foutmelding weergegeven;

- tijdens de ontwikkeling van een programma willen we soms dat een deel van de programmatekst nog niet wordt gecompileerd. In deze programmatekst kunnen ook reeds commentaarteksten voorkomen. Dank zij het nesten van commentaar kunnen we steeds een deel van de programmatekst tot het commentaar rekenen.

```
(* begin tot het commentaar te rekenen tekst
x := 3;
(* dit commentaar beëindigt de vorige niet *)
...
y := 4;

einde tot het commentaar te rekenen tekst  *)
```

Wanneer op deze wijze stukken tekst van compilatie worden uitgesloten, is het voor de lezer soms moeilijk uit te maken of bepaalde opdrachten al of niet tot het programma behoren. Voor de duidelijkheid kunnen we voor het begin van elke tot het commentaar te rekenen regel een bijzonder kenmerk plaatsen :

```
(* begin tot het commentaar te rekenen tekst
VERWIJDERD ...
VERWIJDERD x := 3;
VERWIJDERD y := 4;
einde tot het commentaar te rekenen tekst *)
```

- indien de commentaartekst zelf het begin- of eindsymbool voor commentaar bevat, dan moeten we zelf voor evenveel openings- als sluitcommentaarhaakjes zorgen :

```
(*(* dit commentaar bevat het eindsymbool '*)' voor commentaar *)

(* dit commentaar bevat het beginsymbool '(*' voor commentaar *)*)
```

Literatuur

Bondy J., 'Outline of Draft Modula-2 Documentation, Naming Styles',
Modula-2 News 1, januari 1985

Caillieu R., 'How to Avoid to Get SCHLONKED by Pascal',
Sigplan Notices, V17 # 12, december 1982

Hoppe J., 'Some Problems with the Specification of Standard Modules',
Modula-2 News Ø, oktober 1984

Pentzlin K. L., 'Syntax of Comments : Discussion and Proposal',
Sigplan Notices, V17 # 11, november 1982

Scowen R. S., 'A Standard Syntactic Metalanguage',
Sigplan Notices

Spector D., 'Lexing and Parsing Modula-2',
Sigplan Notices, V18 # 1Ø, oktober 1983

Weinmann J. B., 'Nestable Bracketed Comments',
Sigplan Notices, V18 # 1Ø, oktober 1983

Oefeningen

1. Welke van de volgende stringconstanten zijn juist ?

a. "12 34 56 78 9Ø"
b. "2*a = 2*b + 2*c"
c. '"Hello"'
d. "Het antwoord is "juist""
e. 'Antwoord "J" of "N"'
f. "Ø 1 2 3 4 5 6 7 8 9'

2. Wat is het resultaat van de volgende programmamodule ?

```
MODULE Hello;
(* Programma voor de echo van de invoer
FROM SimpleIO IMPORT ReadString, WriteString;

VAR s : ARRAY [Ø..1Ø] OF CHAR;

BEGIN
ReadString(s);
WriteString(s)
END Hello. *)
```

3 Elementaire programma-objecten

Een programma is een werkvoorschrift dat bij de uitvoering leidt tot bewerkingen op gegevens. De gegevens worden in het programma voorgesteld door objecten. Aan de meeste objecten verbinden we de volgende eigenschappen : een identifier, een type en een waarde. Variabelen zijn objecten waarvoor de waarde tijdens de uitvoering van het programma mag wijzigen. Constanten zijn objecten waarvoor de initiële waarde behouden blijft. Het type van een object definieert de verzameling waarden die we aan het object kunnen toekennen en de bewerkingen die we met het object verrichten.
In dit hoofdstuk behandelen we eerst de eigenschappen van enkelvoudige, scalaire, typen. Daarna beschrijven we de declaratie van objecten en tot slot de toekenningsopdracht.

3.1 Enkelvoudige typen

De enkelvoudige typen zijn de numerieke typen INTEGER, CARDINAL en REAL, het logische type BOOLEAN, het symbooltype CHAR, het enumeratietype en het deelintervaltype.

Het type INTEGER

Dit type stelt een deelverzameling van de verzameling van de gehele getallen voor. Elke waarde van het type INTEGER is dus een geheel getal. De rekenkundige bewerkingen voor objecten van het type INTEGER zijn :

```
+     optellen;
-     aftrekken;
*     vermenigvuldigen;
DIV   geheeltallig delen;
MOD   bepalen van de rest na geheeltallig delen.
```

Het type van het resultaat van deze bewerkingen is opnieuw

INTEGER. Bij geheeltallig delen, aangeduid met DIV, wordt het decimale deel van het quotiënt afgekapt.

```
 15 DIV 4 = 3
-15 DIV 4 = -3
 15 DIV (-4) = -3
```

De operator MOD geeft de rest van de geheeltallige deling van een positief getal door een strikt positief getal. We beschouwen x >= 0 en y > 0 en we definiëren

```
q = x DIV y

r = x MOD y
```

Aan de volgende gelijkheid is steeds voldaan :

```
x = q * y + r
```

waarbij

```
0 <= r < y
```

Ook geldt de eigenschap

```
(x - (x MOD y)) MOD y = 0
```

Voorbeeld :

Uit x = 29, y = 5 volgt

```
q = 29 DIV 5        en        r = 29 MOD 5
  = 5                           = 4
```

Opmerking : De DIV- en MOD-operatoren zijn niet duidelijk gedefinieerd voor negatieve operanden. De tekst in het boek 'Programming in Modula-2' definieert de MOD-operator voor x >= 0 en y > 0, terwijl het rapport 'Report on The Programming Language Modula-2' in hetzelfde boek alleen de beperking y > 0 vermeldt. De implementatie van de MOD-operator kan daarom van compiler tot compiler verschillen.

Behalve deze operatoren, bevat Modula-2 nog de volgende standaard functieprocedures :

ABS(x) : bepaalt de absolute waarde van x;
ODD(x) : geeft het logische resultaat (type BOOLEAN) 'x is oneven'.

en de standaardprocedures :

DEC(x), met x > MIN(INTEGER) : vermindert de waarde van x met 1;

DEC(x,n), met MIN(INTEGER) <= x - n <= MAX(INTEGER) : vermindert de waarde van x met n;

INC(x), met x < MAX(INTEGER) : vermeerdert de waarde van x met 1;

INC(x,n), met MIN(INTEGER) <= x + n <= MAX(INTEGER) : vermeerdert de waarde van x met n.

De verzameling gehele getallen is afhankelijk van de computerimplementatie : bij de meeste computers worden de gehele getallen met type INTEGER voorgesteld door het tweecomplementsysteem. De waardenverzameling is dan beperkt tot het interval

$$-2^{n-1} \,..\, 2^{n-1} - 1$$

waarbij n afhankelijk is van de woordlengte van de computer, meestal 16 of 32. We kunnen de grenzen van dit interval bepalen met de standaardfuncties MIN en MAX. De waarden voor de onder- en bovengrens worden gegeven door :

MIN(INTEGER)

en

MAX(INTEGER)

Indien het resultaat van een rekenkundige bewerking niet tot dit interval behoort, komt overloop voor en is het werkelijke resultaat onbepaald. Over het algemeen wordt de werking van het programma vroegtijdig gestopt.
De procedures DEC en INC zijn niet gedefinieerd voor de grenswaarden van het interval. DEC(x, n) met n > 0 is niet gedefinieerd als x - n < MIN(INTEGER). Ook is INC(x, n) met n > 0 niet gedefinieerd als x + n > MAX(INTEGER). Deze bewerkingen geven, afhankelijk van de implementatie, meestal aanleiding tot een fout in het bereik van het type (range error).

Het type CARDINAL

Het type CARDINAL stelt de verzameling natuurlijke getallen voor. Voor het type CARDINAL gelden dezelfde operatoren en standaardprocedures - behalve de procedure ABS - als voor het type INTEGER.

We gebruiken het type CARDINAL voor een object als we duidelijk weten dat aan het object geen negatieve waarden hoeven te worden toegekend. Bij een eventuele toekenning van een negatieve waarde wordt het programma onderbroken.

Voor de voorstelling van gehele getallen gebruikt een computer n bits. Het bereik voor de natuurlijke getallen wordt nu

$$0 .. 2^n - 1$$

terwijl de grootste waarde van het type INTEGER beperkt is tot

$$2^{n-1} - 1.$$

De grenswaarden voor het type CARDINAL worden gegeven door :

MIN(CARDINAL) = Ø

en

MAX(CARDINAL)

We kunnen aan een object van het type CARDINAL de waarde toekennen van een object van het type INTEGER onder de voorwaarde dat deze waarde behoort tot het domein van beide objecten. Ook de omgekeerde eigenschap geldt. De typen CARDINAL en INTEGER zijn overdraagbaar voor toekenningsopdrachten.

Type-overdrachtfunctie

Het gebruik van objecten van het type CARDINAL en INTEGER in dezelfde rekenkundige uitdrukking is niet toegestaan : Modula-2 verhindert elke gemengde uitdrukking. We lossen dit probleem op door het gebruik van 'type-overdrachtfunctie'. Een type-overdrachtfunctie wijzigt het type van het argument. De naam van elk type geldt ook voor de overdrachtfunctie.

Voorbeeld : het type van i is INTEGER, het type van c is CARDINAL. De gemengde uitdrukking

i + c

is niet toegelaten. Met de overdrachtfunctie INTEGER(c) is de uitdrukking

i + INTEGER(c)

geldig. Analoog geldt de uitdrukking

CARDINAL(i) + c

Het gebruik van een typenaam als een overdrachtfunctie is in Modula-2 algemeen, zelfs voor zelfgedefinieerde typen. Een type-overdrachtfunctie zet het type van het argument om in het type van de functie-identifier : het argument wordt geïnterpreteerd volgens de eigenschappen van de functie-identifier. Een overdrachtfunctie wijzigt dus de betekenis van het argument.
Een type-overdrachtfunctie genereert geen machinecode maar geeft meer vrijheid in het controleren van typen tijdens de compilatie. De type-overdrachtfuncties zijn slechts zinvol voor objecten waaraan dezelfde geheugenruimte wordt toegewezen.

Het type REAL

Waarden van het type REAL vormen een deelverzameling van de reële getallen. Voor objecten van het type REAL zijn ook de rekenkundige basisbewerkingen gedefinieerd. Het symbool voor de delingsoperator is '/'. Daarnaast zijn de prefixoperator '-' en de bewerking ABS beschikbaar.
Een constante van het type REAL bevat steeds een decimale punt en eventueel een schaalfactor. Enkele voorbeelden van REAL-constanten zijn

1.2 1.20 0.0

De schaalfactor is samengesteld uit de letter 'E' en een geheel getal. De factor betekent dat het voorafgaande getal moet worden vermenigvuldigd met een macht van 10 met de exponent gelijk aan de schaalfactor.

1.4E3 betekent 1400.0
2.34E-2 betekent 0.0234

De waarden voor het type REAL worden inwendig voorgesteld door een combinatie (fractie, schaalfactor), de drijvende-komma-voorstelling. Uiteraard worden voor beide onderdelen een beperkt aantal cijferposities gebruikt. Het gevolg hiervan is dat reële waarden niet exact worden voorgesteld en dat de bewerkingen op deze benaderde waarden grote onnauwkeurigheden kunnen veroorzaken. De waarden van het type REAL liggen tussen MIN(REAL) en MAX(REAL).

Conversie

De waarden van het type CARDINAL worden op een totaal verschillende manier voorgesteld dan de waarden van het type REAL. Gemengde uitdrukkingen, waarbij operanden van een verschillend type voorkomen, zijn niet toegelaten. De inwendige voorstelling

van de waarde van een object van het type CARDINAL kan worden omgezet in de (benaderde) waarde van het type REAL met de functie FLOAT. De functie

FLOAT(c)

vormt de waarde van c om tot de drijvende-komma-voorstelling voor c. De functie

TRUNC(r)

met 0 <= r < MAX(CARDINAL) + 1, vormt de waarde van r om tot een natuurlijk getal met type CARDINAL. Het decimale deel wordt afgekapt. Het type van r is REAL.

Een conversiefunctie zet het argument om in een ander type. Bij deze omzetting wordt de betekenis van het argument zo goed mogelijk behouden.

Rekenkundige uitdrukkingen

Een uitdrukking is in het algemeen samengesteld uit verschei- dene operanden die van elkaar worden gescheiden door operatoren. Eventueel kunnen ook ronde haakjes voorkomen. Als operanden kun- nen voorkomen : constanten, variabelen en functies. De functies worden in een later hoofdstuk beschreven. Elke uitdrukking is van een bepaald type : alle operanden in de uitdrukking moeten tot dit type te behoren. Voor de berekening van de waarde wordt voor elke operand de lopende waarde genomen. Elk object in de uitdrukking moet dus reeds een waarde hebben. De waarde van de uitdrukking is ondubbelzinnig bepaald omdat tijdens de evaluatie de volgende regels worden gevolgd :

- de bewerkingen met een hogere prioriteit worden steeds eerst uitgevoerd. Voor de rekenkundige operatoren gelden de volgende prioriteitsregels :

    hoogste prioriteit : '*' '/' 'DIV' 'MOD'
    laagste prioriteit : '+' '-'

- de bewerkingen met gelijke prioriteit worden van links naar rechts uitgevoerd;

- de haakjes '(' en ')' kunnen deze eerste regels doorbreken : een uitdrukking tussen deze haakjes wordt eerst geëvalueerd volgens de gegeven regels.

Enkele voorbeelden :

```
2 * 3 + 4 * 5         = (2 * 3) + (4 * 5) = 26
15 DIV 4 * 3          = (15 DIV 4) * 3    = 9
15 DIV (4 * 3)        = 15 DIV 12         = 1
2 + 3 * 3 - 5         = 2 + (3 * 3) - 5   = 6
10.0 / 2.0 * 5.0      = (10.0/2.0) * 5.0  = 25.0
10.0 / (2.0 * 5.0)    = 10.0 / 10.0       = 1.0
```

Een uitdrukking bestaat uit opeenvolgende termen. De uitdrukking

$$T_1 + T_2 + ... + T_n$$

is equivalent met de uitdrukking

$$((T_1 + T_2) + ...) + T_n.$$

De syntaxisregels voor deze uitdrukking zijn :

```
OptelOperator = '+' | '-'
EnkelvoudigeUitdrukking = ['+' | '-'] term {OptelOperator term}
```

Elke term bestaat uit opeenvolgende factoren. De term

$$F_1 * F_2 * ... * F_n$$

is equivalent met de term

$$((F_1 * F_2) * ...) * F_n.$$

De syntaxisregels voor de term zijn :

```
VermenigvuldigOperator = '*' | '/' | 'DIV' | 'MOD'
term = factor {VermenigvuldigOperator factor}
```

Elke factor is een constante, een variabele, een functie of een uitdrukking tussen haakjes.

Opmerking : in een uitdrukking mogen nooit twee operatoren onmiddellijk naast elkaar voorkomen. De uitdrukking 'a * - b' is fout en moeten we noteren als 'a * (- b)'.

Het type BOOLEAN

Een waarde van het type BOOLEAN is één van de waarden die worden aangegeven door de standaardidentifiers TRUE en FALSE. De identifier voor een boole variabele is meestal een adjectief :

gevonden, aanwezig, foutief

Een waarde TRUE impliceert de aanwezigheid van de eigenschap, de waarde FALSE de afwezigheid. Op de objecten van het type BOOLEAN is een verzameling logische operatoren gedefinieerd : AND, OR en NOT. Voor de operatoren AND en NOT worden respectievelijk ook de symbolen '&' en '~' (tilde) gebruikt.

Logische uitdrukkingen

Met de operatoren en objecten van het type BOOLEAN stellen we logische (of boole) uitdrukkingen samen. Een logische uitdrukking is samengesteld uit termen. De uitdrukking

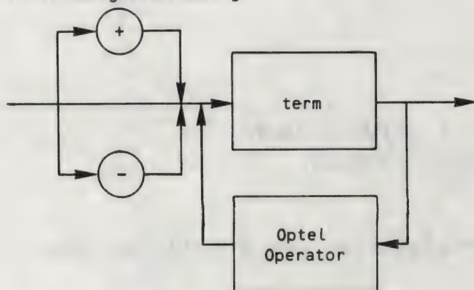
$$T_1 \text{ OR } T_2 \text{ OR } \ldots \text{ OR } T_n$$

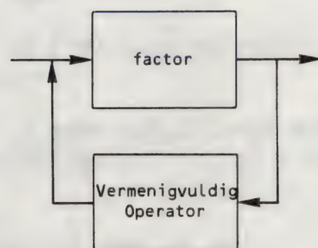
is equivalent met de uitdrukking

$$((T_1 \text{ OR } T_2) \text{ OR } \ldots) \text{ OR } T_n.$$

De syntaxisregels voor deze uitdrukking zijn :

```
OptelOperator = 'OR'
EnkelvoudigeUitdrukking = term {OptelOperator term}
```





Elke term bestaat uit opeenvolgende factoren. De term

$$F_1 \text{ AND } F_2 \text{ AND } \ldots \text{ AND } F_n$$

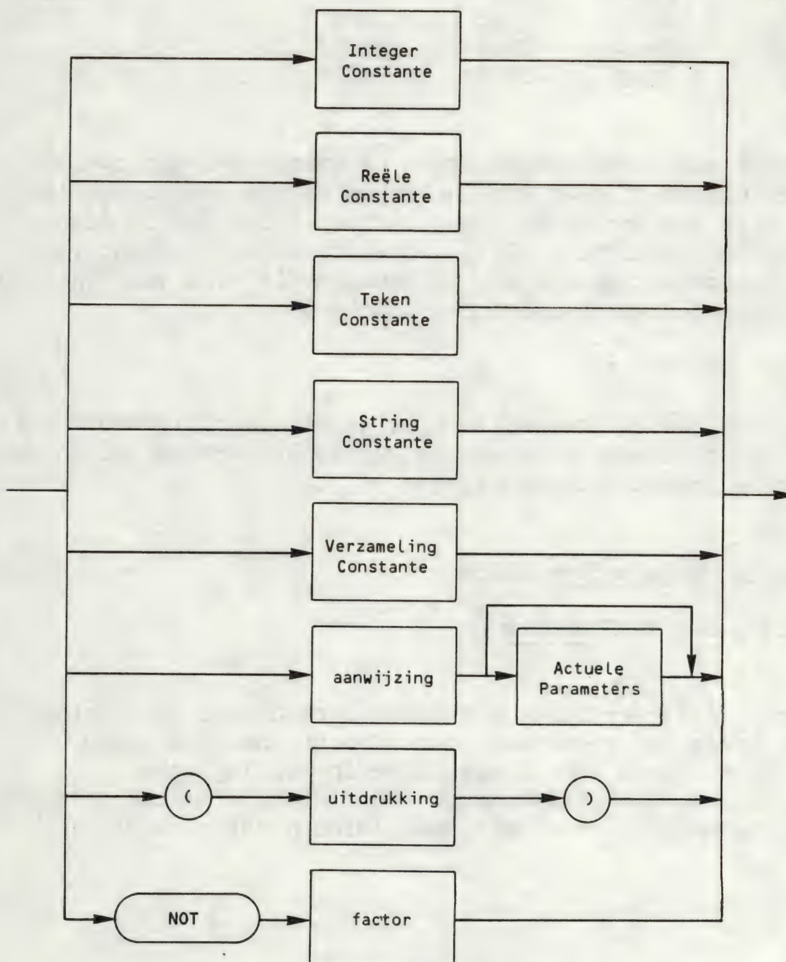
is equivalent met de term

$$((F_1 \text{ AND } F_2) \text{ AND } \ldots) \text{ AND } F_n.$$

De syntaxisregels voor de term zijn :

```
VermenigvuldigOperator = 'AND' | '&'
term = factor {VermenigvuldigOperator factor}
```

Een factor wordt vervangen door een constante, een variabele, een functie of een uitdrukking tussen haakjes. Eventueel wordt een factor voorafgegaan door de NOT-operator.



Voorbeelden :

p OR q
s AND t AND r
u AND (x OR y)

Voor de boole operatoren geldt de volgende hiërarchie : de NOT-operator heeft de hoogste prioriteit, gevolgd door de AND-operator. De OR-operator heeft de laagste prioriteit. Met behulp van de haakjes kunnen we de associaties van een operator met zijn operanden explicieter maken.
De betekenis van deze operatoren wordt gegeven in de volgende waarheidstabel :

p	q	p AND q	p OR q	NOT p
FALSE	FALSE	FALSE	FALSE	TRUE
FALSE	TRUE	FALSE	TRUE	TRUE
TRUE	FALSE	FALSE	TRUE	FALSE
TRUE	TRUE	TRUE	TRUE	FALSE

Het is mogelijk dat in een uitdrukking de waarde van een operand niet is gedefinieerd of niet kan worden berekend. Bijvoorbeeld, het element a_{101} van een lijst a met indices 1 tot 100 is niet gedefinieerd; het quotiënt x DIV y kan niet worden berekend voor y = 0. Ook het resultaat van een uitdrukking is niet gedefinieerd als hierin ongedefinieerde operanden voorkomen :

(y # 0) AND (x DIV y = z)

De waarde van de tweede operand is niet gedefinieerd indien y = 0. In Modula-2 wordt de exacte betekenis van de operatoren AND en OR gegeven door de logische uitdrukkingen :

p AND q = <u>als</u> p <u>dan</u> q <u>anders</u> FALSE

p OR q = <u>als</u> p <u>dan</u> TRUE <u>anders</u> q

Deze definitie impliceert dat de tweede operand niet moet worden geëvalueerd indien het resultaat van de eerste operand reeds bekend is. De volgorde van de operanden in een logische uitdrukking is dus niet willekeurig. Voor deze definitie geldt de volgende waarheidstabel ('—' betekent 'niet gedefinieerd') :

p	q	p OR q	p AND q
FALSE	FALSE	FALSE	FALSE
FALSE	TRUE	TRUE	FALSE
FALSE	--	--	FALSE
TRUE	FALSE	TRUE	FALSE
TRUE	TRUE	TRUE	TRUE
TRUE	--	TRUE	--
--	FALSE	--	--
--	TRUE	--	--
--	--	--	--

Opmerking : in tegenstelling tot de rekenkundige operatoren, mogen in een logische uitdrukking de operatoren AND, OR en NOT onmiddellijk voor de operator NOT voorkomen. Bijvoorbeeld :

```
a AND NOT b AND c    betekent   (a AND (NOT b)) AND c
x OR NOT y AND z     betekent   x OR ((NOT y) AND z)
NOT NOT x            betekent   NOT (NOT x)  of  x
```

Condities

Op de waardenverzameling van de typen INTEGER, CARDINAL en REAL is een volgorde gedefinieerd. Deze typen noemen we scalaire typen. Een conditie is de toepassing van een relationele operator op twee operanden van het scalaire type. De relationele operatoren waarover we kunnen beschikken zijn :

=	gelijk aan
<>	niet gelijk aan
<	kleiner dan
<=	kleiner dan of gelijk aan
>	groter dan
>=	groter dan of gelijk aan

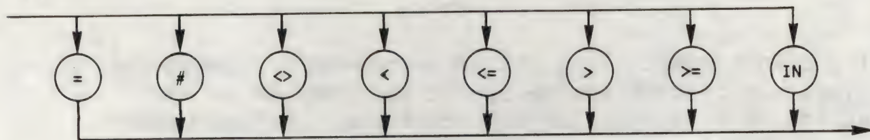
Het symbool '#' is een synoniem van '<>'. Een conditie bevat twee operanden. De twee operanden worden gescheiden door een relationele operator en behoren tot hetzelfde type. Het type van het resultaat van een conditie is BOOLEAN. Bijvoorbeeld :

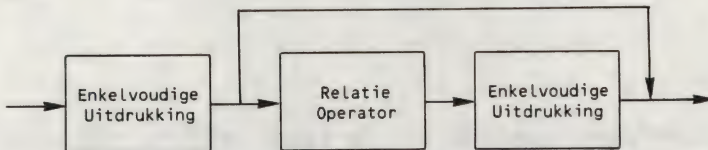
```
'a' = 'A'                  FALSE
 5  <  9                   TRUE
("a" # "b") AND (6 < 9)    TRUE
```

De syntaxis voor de uitdrukking van een conditie is :

```
uitdrukking = EnkelvoudigeUitdrukking
              [RelatieOperator EnkelvoudigeUitdrukking]

RelatieOperator = '=' | '#' | '<>' | '<' | '<=' | '>' | '>='
                  | 'IN'
```





Enkele voorbeelden :

```
x = y
(1Ø <= x) AND (x <= 1ØØ)
(x > y) OR (y = Ø)
```

We mogen de relationele operatoren ook toepassen op boole waarden. Voor deze waarden geldt de eigenschap

FALSE < TRUE

We passen dit toe op de logische implicatie, waarvoor de waarden worden weergegeven in de waarheidstabel :

p	q	p impliceert q
TRUE	TRUE	TRUE
TRUE	FALSE	FALSE
FALSE	TRUE	TRUE
FALSE	FALSE	TRUE

In Modula-2 drukken we de logische implicatie p => q uit als

NOT p OR q

of

p <= q

Het type CHAR

Een waarde van het type CHAR is een teken dat behoort tot de verzameling van alle tekens die in het systeem, waarop het programma draait, beschikbaar is. De ASCII-verzameling (American Standard Code for Information Interchange) is de Amerikaanse standaard en wordt het meest toegepast. De verzameling van de tekens is geordend : elk teken heeft een vaste plaats in de verzameling. Met deze plaats stemt een volgnummer overeen. De ASCII-verzameling wordt in de volgende tabel afgebeeld. Het volgnummer van een teken is de som van het kolomnummer en het rijnummer. Deze nummers worden in Modula-2 meestal in octale notatie geschreven.

octaal rij-nummer : \ **octaal kolomnummer :**	Ø	2Ø	4Ø	6Ø	1ØØ	12Ø	14Ø	16Ø
Ø	nul	dle		Ø	@	P		p
1	soh	dc1	!	1	A	Q	a	q
2	stx	dc2	"	2	B	R	b	r
3	etx	dc3	#	3	C	S	c	s
4	eot	dc4	$	4	D	T	d	t
5	enq	nak	%	5	E	U	e	u
6	ack	syn	&	6	F	V	f	v
7	bel	etb	'	7	G	W	g	w
1Ø	bs	can	(	8	H	X	h	x
11	ht	em	)	9	I	Y	i	y
12	lf	sub	*	:	J	Z	j	z
13	vt	esc	+	;	K	[	k	{
14	ff	fs	,	<	L	\	l	|
15	cr	gs	-	=	M	]	m	}
16	so	rs	.	>	N	↑	n	~
17	si	us	/	?	O	←	o	del

De eerste twee kolommen bevatten de besturingstekens, aangeduid met de gebruikelijke afkortingen. De betekenis van deze besturingstekens wordt in hoofdzaak bepaald door de interpretatie ervan in een programma. Deze tekens zijn meestal niet afdrukbaar.

De (afdrukbare) waarde van een object van het type CHAR wordt aangeduid door omsluiting van het teken met enkele (') of dubbele (") aanhalingstekens.

'a' "a" '"' "'"

De waarde van een niet-afdrukbaar teken kunnen we niet op deze wijze specificeren. De waarde van een teken wordt ook gegeven door een numerieke tekenconstante : het octale volgnummer gevolgd door de letter 'C'.

14C voor ff (form feed)

12C voor lf (line feed)

Opmerking : Op talrijke computersystemen is het type CHAR gedefinieerd met acht bits. De ordinaalwaarden Ø tot 127 stemmen overeen met de ASCII-code. De overige waarden zijn niet gestandaardiseerd. Zij worden bijvoorbeeld gebruikt voor diakritische of samengestelde tekens.

Overdrachtfuncties

Een waarde van het type CHAR kan uitsluitend aan objecten van het type CHAR worden toegekend. Rekenkundige bewerkingen op het volgnummer zijn toegelaten na het gebruik van de overdrachtfunctie

ORD(teken)

Deze functie levert het volgnummer van een teken af. Het type van het volgnummer is CARDINAL.

ORD('A') geeft 65

ORD('Ø') geeft 48

Omgekeerd levert de functie

CHR(n) met MIN(CHAR) <= n <= MAX(CHAR)

het teken met volgnummer n.

CHR(65) geeft 'A'

Voor de functies ORD en CHR gelden :

ORD(CHR(i)) = i

en

CHR(ORD(teken)) = teken.

De functie VAL is ook gedefinieerd voor het type CHAR. De betekenis is dezelfde als de functie CHR :

VAL(CHAR,i) = CHR(i)

Met behulp van de ORD-functie definiëren we de ordening van de tekens. Voor de verzameling tekens geldt bijvoorbeeld :

ORD("A") = ORD("B") - 1
ORD("B") = ORD("C") - 1

We beschouwen C_1 en C_2 als constanten van het type CHAR en de relatie R ('<', '<=', '>=', '>', '#' of '='). Dan is

C_1 R C_2

equivalent met

ORD(C_1) R ORD(C_1)

Voorbeeld :

We berekenen de numerieke waarde van een cijfer met

ORD(cijfer) - ORD("Ø")

en we bepalen het cijfer voor een numerieke waarde n, met Ø <= n <= 9, met

CHR(n + ORD("Ø"))

Andere standaardprocedures

Voor het type CHAR zijn ook procedures DEC en INC gedefinieerd. Daarnaast is de functie CAP beschikbaar. De betekenis van deze procedures is :

DEC(x), met x > MIN(CHAR) : de waarde van de variabele x wordt vervangen door de voorganger van x in de verzameling tekens;

DEC(x,n), met ORD(MIN(CHAR)) <= ORD(x) - n <= ORD(MAX(CHAR)) : de waarde van de variabele x wordt vervangen door de n-de voorganger van x in de verzameling tekens;

INC(x), met x < MAX(CHAR) : de waarde van de variabele x wordt vervangen door de opvolger van x in de verzameling tekens;

INC(x,n), met ORD(MIN(CHAR)) <= ORD(x) + n <= ORD(MAX(CHAR)) : de ..aarde van de variabele x wordt vervangen door de n-de opvolger van x in de verzameling tekens.

CAP(x) : indien x behoort tot de verzameling kleine letters {"a" .. "z"} dan wordt x vervangen door de overeenkomstige hoofdletter. Voor alle andere tekens heeft deze functie geen effect.

3.2 Het declareren van identifiers

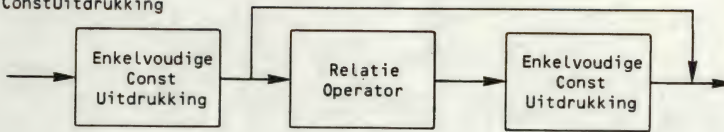
De objecten die in een programma worden bewerkt, worden ingedeeld in constanten en variabelen. De waarde van een constante wijzigt niet tijdens de verwerking van een programma. De waarde van een variabele kunnen we eventueel wijzigen met een toekenningsopdracht of met een leesopdracht. Elk object is van een bepaald type. Het type van een constante is afhankelijk van de waarde. Een type wordt expliciet aan een variabele verbonden door een declaratie. Naast het type wordt in de declaratie tevens de identifier voor een variabele vastgelegd. Eventueel kunnen we ook met een 'constantedeclaratie' een identifier aan een constante toekennen. De regels voor het specificeren van deze declaraties worden in de volgende paragrafen beschreven.

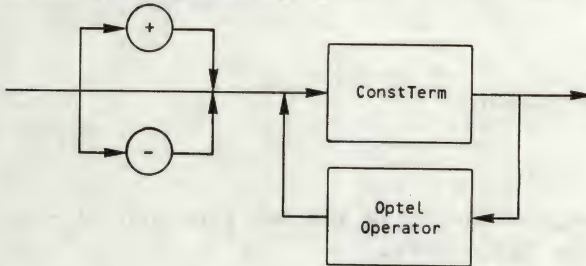
Constantedeclaraties

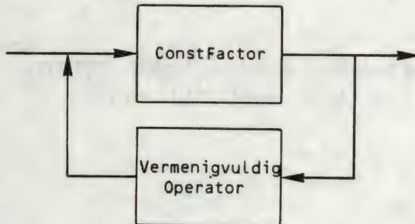
Een constante in een programma geven we aan door de waarde te schrijven op de plaats waar de constante nodig is. Voor het onderhoud en de leesbaarheid van programma's is deze schrijfstijl zeer nadelig : de betekenis van de constanten in de tekst gaat verloren en bij wijzigingen moeten we de waarden op veel plaatsen in de tekst aanpassen. Het gevaar is groot dat hier en daar aanpassingen worden vergeten. In een goed opgesteld programma worden de constante gegevenselementen in constantedeclaraties geïsoleerd. In een constantedeclaratie geven we aan elke constante een naam. Daarna is iedere verschijning van de naam in de programmatekst equivalent met de waarde van de constante. De syntaxis voor een constantedeclaratie is :

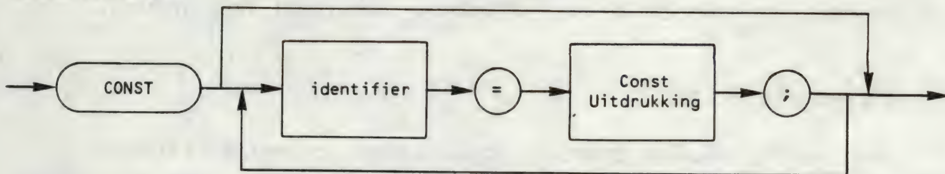
```
ConstUitdrukking = uitdrukking
ConstDeclaratie  = identifier '=' ConstUitdrukking
```

Een ConstUitdrukking komt syntactisch overeen met een EnkelvoudigeUitdrukking waarvan de waarde tijdens de vertaling kan worden berekend. Een opeenvolging van constantedeclaraties wordt voorafgegaan door het symbool 'CONST'. Opeenvolgende declaraties worden met een ';' van elkaar gescheiden.


ConstUitdrukking
Enkelvoudige Const Uitdrukking
Relatie Operator
Enkelvoudige Const Uitdrukking


EnkelvoudigeConstUitdrukking
+
-
ConstTerm
Optel Operator


ConstTerm
ConstFactor
Vermenigvuldig Operator


ConstDeclaratie
CONST
identifier
=
Const Uitdrukking
;

Voorbeeld :

```
CONST n         = 100;
      maxAantal = n * 2;
      s         = 'Modula-2';
      EOL       = 36C;
```

In een constantedeclaratie kunnen ook standaard procedurefuncties worden gebruikt indien de waarde tijdens de compilatie kan worden berekend. Bijvoorbeeld de declaraties

```
CONST nul    = ORD('0');
      n      = 100;
      grens  = MAX(CARDINAL) - n;
```

zijn toegelaten.

Het type van de constante is gelijk aan het type van de uitdrukking die de constante definieert.

Voorbeeld :

```
CONST endStr  = 0C;    (* type CHAR *)
      min1    = -1;    (* type INTEGER *)
```

Als de waarde van de uitdrukking behoort tot verschillende typen, behoort ook het type van de constante tot deze verschillende typen.

Voorbeeld :

```
CONST nul     = 0;
      aantal  = 1000;
```

Het type van aantal en nul is INTEGER, CARDINAL of elk deelintervaltype met basistype INTEGER of CARDINAL. Het deelintervaltype wordt in een volgende paragraaf beschreven.

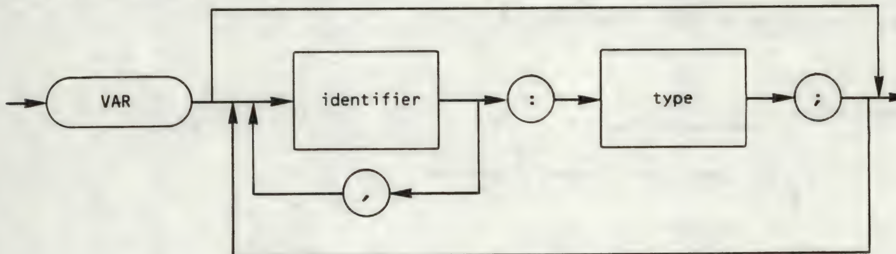
Variabeledeclaraties

Variabelen worden steeds gedefinieerd in een declaratie voordat we de variabele in een opdracht gebruiken. Een variabele-declaratie verbindt een identifier en een type met een object. Na de declaratie is de waarde van de variabele nog niet gedefinieerd. In het programma wordt een waarde toegekend met een toekennings- of een leesopdracht.

De syntaxis voor een variabeledeclaratie is :

```
IdentLijst           = identifier { ',' identifier }
VariabeleDeclaratie = IdentLijst ':' type
```

Variabelen van een zelfde type mogen we opsommen in een enkele lijst. Een opeenvolging van variabeledeclaraties wordt voorafgegaan door het symbool 'VAR'. De declaraties worden gescheiden met een ';'.



Voorbeelden :

```
VAR i, j, k : CARDINAL;
    letter  : CHAR;
    x, y    : REAL
```

Typedeclaraties

De typen INTEGER, CARDINAL, REAL, CHAR en BOOLEAN worden in hoofdzaak gebruikt voor de bewerking van numerieke, logische en op tekst georiënteerde gegevens. Deze standaardtypen kunnen we zonder voorafgaande definitie in een programma gebruiken. Voor de oplossing van veel problemen verschaffen deze typen niet de waarden die zijn aangepast aan de objecten van het probleem : bijvoorbeeld de dagen van de week, de bewerkingen op een bestand, de foutcondities voor een bepaalde gegevensverwerking. Voor deze toepassingen definiëren we een enumeratietype : het bevat de opsomming van alle waarden die tot het type behoren.
Voor andere toepassingen is het domein van de waarden die we aan het object toekennen beperkt tot een interval van een gegeven type. Bijvoorbeeld de jaartallen van de twintigste eeuw behoren tot het interval [1900..1999]. Voor dit type probleem definiëren we een deelintervaltype.

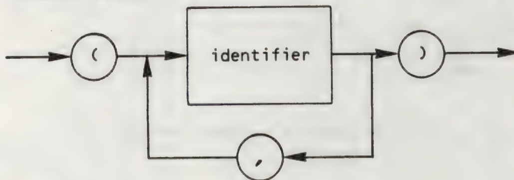
Het enumeratietype

De typedeclaratie

$$T = (C_1,C_2,\ldots,C_n)$$

definieert het type T met de waarden C_1, C_2 tot C_n. Elk van deze waarden wordt aangeduid met een constante-identifier. Deze identifiers zijn de enige waarden van het type.
De syntaxis voor het enumeratietype is :

```
IdentLijst     = identifier { ',' identifier }
EnumeratieType = '(' IdentLijst ')'
```



Voorbeeld :

De dagen van de week definiëren we met

```
Weekdag = (maandag, dinsdag, woensdag, donderdag,
           vrijdag, zaterdag, zondag)
```

In een voorstel voor de standaardisering van de Modula-2 modulebibliotheek wordt de toegang tot een bestand gegeven door

```
ReadWriteMode = (readOnly, readWrite, appendOnly)
```

en de toestand waarin een bestand zich bevindt door

```
FileState = (ok, nameError, noFile, existingFile, deviceError,
             noMoreRoom, accessError, notOpen, endError,
             outsideFile, otherError).
```

Overdrachtfuncties

Met elke identifier van een enumeratietype komt een volgnummer overeen. We verkrijgen dit volgnummer met de standaardfunctie ORD. Voor een identifier C_n geldt :

$$ORD(C_n) = n - 1$$

Bijvoorbeeld :

ORD(ok) = Ø, ORD(appendOnly) = 2

Ook de inverse functie VAL geldt voor het enumeratietype. De algemene vorm voor deze functie is

VAL(TypeIdentifier, volgnummer)

en betekent

VAL(TypeIdentifier, ORD(x)) = x.

Zo levert de functie VAL(Weekdag, 3) de waarde 'donderdag'.

Condities

De relationele operatoren kunnen ook op de enumeratietypen worden toegepast. Het type van het resultaat van een dergelijke conditie is natuurlijk BOOLEAN.

ok < otherError levert TRUE

De conditie

maandag < dinsdag levert TRUE

kunnen we interpreteren als 'maandag komt voor dinsdag'.

Het type BOOLEAN is te beschouwen als het enumeratietype

BOOLEAN = (FALSE, TRUE)

waarvoor geldt

FALSE < TRUE.

Andere standaardprocedures

Met de standaardprocedures DEC en INC bepalen we de voorganger en de opvolger van de huidige waarde die aan een object met een enumeratietype is toegekend.

Voorbeeld :

Beschouw de variabele x van het type FileState; de huidige waarde van x is 'nameError'. Met de procedure

DEC(x) krijgt x de waarde 'ok'

en met

INC(x) krijgt x de waarde 'noFile'

Voor de grenswaarden van een enumeratietype is het resultaat van deze procedures niet gedefinieerd. Beschouw de opsomming

S = (x, y, z)

en een variabele v van het type S. De huidige waarde van v is x. Dan is het resultaat :

DEC(v) onbepaald;
INC(v) v krijgt de waarde y

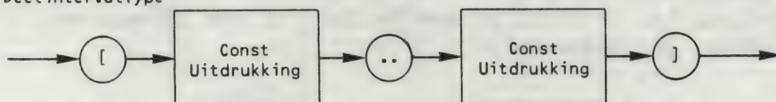
Het deelintervaltype

Wanneer de waarden van een object behoren tot een interval van de waarden van een enkelvoudig type, dan definiëren we hiervoor een deelintervaltype. Het deelinterval wordt gedefinieerd door een onder- en een bovengrens. Beschouw een type T waarvoor de waarden begrepen zijn tussen 1 en N. Hiervoor specificeren we

T = [1..N]

Elk deelintervaltype heeft een basistype. De bewerkingen op het basistype zijn ook toepasbaar op het deelintervaltype. De enige beperking is de verzameling waarden die we aan een object met een deelintervaltype kunnen toekennen. De syntaxis voor het deelintervaltype is

```
DeelintervalType = [kwalificatie]
                   '[' ConstUitdrukking..ConstUitdrukking ']'
```



De eerste ConstUitdrukking specificeert de ondergrens, de tweede de bovengrens voor de waarden van het type. De uitdrukkingen mogen alleen constanten bevatten. Voor de waarden van de onder- en bovengrens geldt steeds

ondergrens <= bovengrens

Het type van de onder- en bovengrens noemen we het basistype van het deelinterval.

Voorbeeld :

De logische schijfaandrijvingen van een microcomputersysteem worden beschreven met het type

```
Aandrijving = ["A".."O"]
```

De hoofdletters definiëren we met

```
Hoofdletter = ["A".."Z"]
```

en een werkdag met

```
Werkdag = [maandag..vrijdag]
```

Voor een jaartal definiëren we

```
Jaartal = [1980..2000]
```

Het basistype voor een deelinterval wordt afgeleid van de grenzen van het interval. Voor de voorbeelden is het basistype :

Aandrijving	CHAR
Hoofdletter	CHAR
Werkdag	Weekdag

Het basistype voor 'Jaartal' kunnen we niet ondubbelzinnig afleiden : zowel het type INTEGER als het type CARDINAL komen in aanmerking. Voor de gehele getallen geldt de volgende regel : als de ondergrens negatief is, dan is het basistype INTEGER. In het andere geval is het basistype CARDINAL. Om elke dubbelzinnigheid te vermijden kunnen we het basistype vermelden in de definitie met de kwalificatie INTEGER :

```
Jaartal = INTEGER[1980..2000]
```

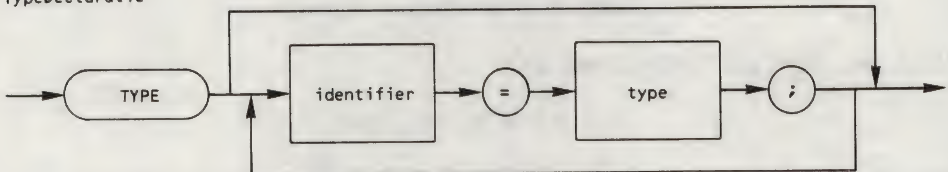
Een object van het type 'Jaartal' kunnen we nu zonder overdrachtfuncties in een INTEGER-uitdrukking gebruiken.

Samenvatting syntaxis voor typedeclaraties

In dit hoofdstuk beschreven we uitsluitend de declaraties van elementaire programma-objecten. Voor de typedeclaraties is het enumeratietype en het deelinterval gedefinieerd. Andere typen worden in de volgende hoofdstukken toegevoegd. De voorlopige syntaxis voor een typedeclaratie is :

```
TypeDeclaratie    = identifier '=' type
type              = EnkelvoudigType
EnkelvoudigType   = kwalificatie | EnumeratieType
                    | Deelintervaltype
kwalificatie      = identifier
EnumeratieType    = '(' IdentLijst ')'
DeelintervalType  = [kwalificatie]
                    '[' ConstUitdrukking..ConstUitdrukking ']'
```

Een opeenvolging van typedeclaraties wordt voorafgegaan door het symbool 'TYPE'. De declaraties worden gescheiden met een ';'.



Voorbeeld :

```
TYPE Weekdag = (maandag, dinsdag, woensdag, donderdag, vrijdag,
                zaterdag, zondag);
     Index   = [1..100];
     Jaar    = INTEGER[1900..2000]
```

3.3 Declaratie van programma-objecten

Alle objecten die we in een programma gebruiken, moeten we vooraf declareren. Voor deze declaraties geldt voorlopig de volgende syntaxis :

```
declaraties =   'CONST' {ConstanteDeclaratie}
              | 'TYPE'  {TypeDeclaratie}
              | 'VAR'   {VariabeleDeclaratie}
```

De volgorde van de constante-, type- en variabeledeclaraties is willekeurig. Wanneer we echter in een declaratie D_2 aan een object O_1 refereren, moet de declaratie D_1 van het object O_1 de declaratie D_2 in de tekst voorafgaan. De volgorde van de volgende declaraties is bijvoorbeeld fout :

```
VAR   i           : Index;
CONST bovengrens = 100;
TYPE  Index      = [1..bovengrens];
```

De juiste oplossing is :

```
CONST bovengrens = 100;
TYPE  Index      = [1..bovengrens];
VAR   i          : Index;
```

De syntaxis staat ons wel toe alle declaraties van de objecten die logisch bij elkaar behoren, in de tekst te groeperen, bijvoorbeeld

```
CONST bovengrens = 100;
TYPE  Index      = [1..bovengrens];
VAR   i, j       : Index;

TYPE  Richting   = (links, rechts, boven, onder);
VAR   pijl       : Richting;
```

Opmerking : De declaratie

```
CONST n     = 100;
TYPE  Index = [0..n + 1];
VAR   k     : Index
```

is equivalent met de declaratie

```
CONST n = 100;
VAR   k : [0..n + 1]
```

In de tweede declaratie is het type van de variabele k onbenoemd. Het gebruik van onbenoemde typen heeft dezelfde nadelen als het gebruik van onbenoemde constanten : de leesbaarheid en de onderhoudbaarheid van een programma worden aanzienlijk verminderd. We vermijden daarom zoveel mogelijk onbenoemde typen. Modula-2 moedigt zelfs het gebruik van benoemde typen aan : in de definitie van de parameterlijst van procedures wordt elke parameter met een benoemd type geassocieerd. Een andere vorm is niet toegelaten.

3.4 Operanden met overdraagbaar type

Voor de meeste binaire operatoren moeten de typen van de operanden overdraagbaar zijn. Twee typen T1 en T2 zijn type-overdraagbaar dan en alleen dan als een van de volgende condities waar is :

1. T1 en T2 zijn van hetzelfde type. Dit betekent dat beide worden getypeerd met dezelfde type-identifier :

```
TYPE T1 = CARDINAL;
     T2 = CARDINAL;
```

2. De typen T1 en T2 zijn gelijk gedefinieerd. Dit betekent dat bijvoorbeeld de volgende declaratie is gemaakt :

```
TYPE T1 = type;
     T2 = T1;
```

3. Type T1 is een deelinterval van T2 of omgekeerd. Bij voorbeeld :

```
TYPE T1 = CARDINAL;
     T2 = [1..100];
```

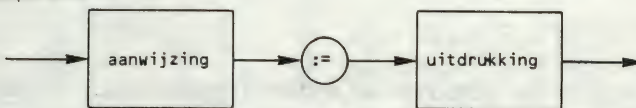
4. T1 en T2 zijn deelintervaltypen van hetzelfde basistype.

```
TYPE T1 = [1..100];
     T2 = [1..200];
```

3.5 De toekenningsopdracht

De eigenschappen van een variabele, het type en de identifier, worden vastgelegd in de variabeledeclaratie. De waarde is door de declaratie niet gedefinieerd. Met een toekenningsopdracht kennen we een waarde toe aan de variabele. De syntaxis voor de toekenningsopdracht is :

ToekenningsOpdracht = identifier ':=' uitdrukking



De uitdrukking aan de rechterzijde van het ':=' symbool wordt eerst geëvalueerd. Daarna wordt de waarde van deze uitdrukking toegekend aan de variabele met de identifier links van het ':='-symbool. Het symbool ':=' betekent 'wordt gelijk aan'.

Voorbeelden :

```
i := 1      de waarde 1 wordt aan de variabele i toegekend;
x := a + b  de som van a en b wordt berekend. De waarde van de
            som wordt toegekend aan de variabele x.
```

Het type van de uitdrukking moet bij de toekenning passen bij het type van de variabele in het linkerlid. Hiervoor gelden de volgende regels :

- het type van de uitdrukking moet passen bij het type van het linkerlid;

- voor het linker- en het rechterlid geldt :

linkerlid	rechterlid
- CARDINAL of basistype = CARDINAL	INTEGER of basistype = INTEGER;
- INTEGER of basistype = INTEGER	CARDINAL of basistype = CARDINAL;
- CHAR	numerieke tekenconstante.

Voorbeelden :

```
VAR teken : CHAR;
    i, j  : CARDINAL;
    a, b  : INTEGER;
```

Voor de volgende toekenningsopdrachten past de waarde van de uitdrukking (rechterlid) bij het linkerlid :

```
teken := "a"
teken := 36C
j := 2
i := j * 2
a := j * 1Ø
b := 5
i := a DIV b
b := i + j
```

Samenvatting

Tot nu toe zijn de scalaire standaardtypen REAL, INTEGER, CARDINAL, BOOLEAN en CHAR behandeld. Deze laatste vier typen, het enumeratietype en het deelintervaltype zijn discrete typen : voor elk van deze typen bestaat een aftelbare verzameling discrete waarden. Voor de discrete typen zijn de bewerkingen DEC, INC, ORD en VAL gedefinieerd. Voor het type CARDINAL betekent dit bijvoorbeeld :

```
DEC(x)   = x - 1
DEC(x,n) = x - n
INC(x)   = x + 1
INC(x,n) = x + n
ORD(x)   = x
VAL(CARDINAL,x) = x
```

De waarden voor de onder- en de bovengrens voor een type worden gegeven door de functies MIN(Type) en MAX(Type). Voor deze grenswaarden zijn de bewerkingen DEC en INC niet gedefinieerd.

Een type-overdrachtfunctie geeft meer vrijheid tijdens de compilatie : een object wordt geïnterpreteerd volgens het type van de overdrachtfunctie. Elk type kunnen we als een overdrachtfunctie gebruiken. Voor deze overdrachtfuncties wordt geen bijkomende machinecode gegenereerd. De functies ORD, CHR en VAL vormen een tweede groep overdrachtfuncties voor de omzetting van en naar de ordinaalwaarde van het argument.

Een conversiefunctie zet de waarde met een gegeven type om in de overeenkomstige waarde van het doeltype. Bij de conversie wordt nieuwe code gegenereerd. De belangrijkste conversiefuncties zijn :

gegeven type	functie	doeltype
REAL	TRUNC	CARDINAL
CARDINAL	FLOAT	REAL

Literatuur

'A Note on the Modulo Operation',
Sigplan Notices, april 1985

R. Bush, 'Modula-2 Library',
Modula-2 News, januari 1985

Oefeningen

1. Gegeven :

```
VAR a : INTEGER;
    b : CARDINAL;
    c : REAL;
```

Spoor de fouten op in de volgende opdrachten :

```
a := 52.3;
b := +10;
c := -16.23;
b := b * c;
c := c / 3;
```

2. Wat is het resultaat van de volgende uitdrukking ? Bepaal ook het type van het resultaat.

```
(TRUNC(23.96) * 3) MOD TRUNC(3.14)
```

3. Wat is het resultaat van de volgende programmamodule ?

```
MODULE t;
FROM SimpleIO IMPORT WriteCard, WriteLn;
VAR c, d : INTEGER;

BEGIN
c := 1;
d := -c;
WriteCard(d, 3);WriteLn
END t.
```

4. Onderzoek de volgende opdrachten : welke opdrachten zijn fout, welke definities gelden voor de MOD- en DIV-operatoren voor je Modula-2 implementatie ?

```
VAR a, b, c, d : INTEGER;

a := 1;
b := 2;

1.  c := a * -b;
2.  d := -a MOD 2;
3.  d := a MOD -2;
4.  c := -b DIV 2;
```

5. In een computersysteem worden de getallen met type INTEGER voorgesteld volgens het tweecomplementsysteem. De woordlengte is zestien bits. Welke waarde krijgt de variabele i met type INTEGER of de variabele j met type CARDINAL bij elk van de volgende opdrachten ?

```
1.  i := ORD(10);
2.  j := ORD(-10);
3.  i := VAL(INTEGER, 10);
4.  i := VAL(INTEGER, 65520);
5.  i := VAL(INTEGER, ORD(10));
6.  j := CARDINAL(-10)
```

6. Welke resultaten worden met de volgende programmamodule afgedrukt ?

```
MODULE t;
FROM SimpleIO IMPORT WriteCard, WriteLn, WriteChar;
VAR c : CARDINAL;

BEGIN
c := 67;
WriteCard(ORD('a'),3); WriteLn;
WriteCard(ORD(CAP('a')),3); WriteLn;
WriteChar(VAL(CHAR, 67)); WriteLn;
WriteChar(VAL(CHAR, c)); WriteLn;
WriteChar(CHAR(67)); WriteLn;
WriteChar(CHR(67)); WriteLn;
END t.
```

7. Wat is het resultaat van de volgende boole uitdrukkingen met

```
VAR a, b : BOOLEAN;
en
a = TRUE
b = FALSE ?
```

```
1. a AND b;
2. NOT a OR b;
3. b AND (6 < 4);
```

8. Bij welke uidrukkingen van oefening 7 wordt de waarde van de volledige uitdrukking uitsluitend bepaald door de eerste operand ?

9. Bepaal het basistype voor de volgende deelintervaltypen :

```
a. Temperatuur = [-20..75];
b. MenuRegel   = [1..10];
c. HexGetal    = ["A".."F"];
```

10. Zoek voor je computersysteem de typen die met een overdrachtfunctie naar elkaar kunnen worden omgezet :

INTEGER	POINTER	CARDINAL
BOOLEAN	CHAR	WORD
REAL	ADDRESS	BITSET

11. Beschouw de declaraties

```
VAR x, y    : INTEGER;
    r, s    : REAL;
    card    : CARDINAL;
    char    : CHAR;
    g       : BOOLEAN;
```

Welke opdrachten zijn niet toegelaten en waarom ?

```
a. x := y * r;
b. x := CARDINAL(s);
c. x := TRUNC(r) * TRUNC(s);
d. r := FLOAT(x * y);
e. g := card;
f. x := TRUNC(r * s);
```

4 Besturingsstructuren

Naast de toekenningsopdracht, die de waarde van een variabele wijzigt, kent Modula-2 ook besturingsstructuren waarmee de volgorde van de acties, behorend bij de opdrachten, wordt vastgelegd. We onderscheiden de drie basisbesturingsstructuren : de sequentiële structuur, de conditionele structuur en de herhalingsstructuur. Voordat we met deze structuren beginnen, worden de standaardprocedure HALT en de RETURN-opdracht beschreven. Met HALT en RETURN kan de verwerking van een programma vroegtijdig worden gestopt.

4.1 HALT en RETURN

Een Modula-2 programma is opgebouwd uit blokken : een programmablok en eventueel een of meer procedureblokken. Elk blok is samengesteld uit twee onderdelen : een declaratiedeel en een actiedeel. In het declaratiedeel worden de programma-objecten gedefinieerd, in het actiedeel worden de opdrachten beschreven. De uitvoering van een blok eindigt bij de laatste opdracht of bij de uitvoering van een RETURN-opdracht. Elk blok kan een of meer RETURN-opdrachten bevatten. We gebruiken de RETURN-opdracht in een blok, of meer specifiek in een programma, als de uitvoering van dit blok vroegtijdig op een normale wijze mag worden beëindigd.

De standaardprocedure HALT beëindigt altijd de uitvoering van het programma. Deze procedure kan op elke willekeurige plaats in een Modula-2 programma worden aangeroepen. We gebruiken de HALT-procedure wanneer een programma op een abnormale wijze eindigt omdat ten gevolge van fouten verdere verwerking zinloos is geworden. Bij de meeste implementaties wordt bij de uitvoering van de HALT-procedure een geheugendump (Engels : memory dump) in een bestand aangemaakt. Dit bestand kan met een symbolisch uitvlooiprogramma (Engels : symbolic debugger) worden geanalyseerd.

4.2 De sequentiële structuur of de opdrachtenrij

Een opdrachtenrij is een aaneenrijging van opdrachten tot een geordende rij acties. Elke opdracht wordt slechts uitgevoerd nadat de vorige is beëindigd. Twee opdrachten uit de rij worden gescheiden door een puntkomma. Als de opdracht S_2 volgt op de opdracht S_1, dan noteren we

$$S_1; S_2$$

De voorlopige syntaxis voor de opdracht is

```
opdracht = [toekenningsopdracht]
```

De opsomming van de opdrachten zullen we in het verloop van de tekst verder aanvullen.

De syntaxis voor een opdrachtenrij is

```
OpdrachtenRij = opdracht { ; opdracht }
```

Voorbeeld :

```
q := Ø; r := r + 1; s := s - q
```

Deze opdrachtenrij bestaat uit de opdrachten

S_1 q := Ø

S_2 r := r + 1

S_3 s := s - q

De acties, die bij deze opdrachten behoren, worden uitgevoerd in de aangegeven volgorde.

De syntaxis van een opdracht impliceert het bestaan van een lege opdracht (Engels : empty statement, null statement). De rij

```
x := y + 1; ; s := s - q
```

bestaat uit drie opdrachten : de toekenningsopdrachten x := y + 1 en s := s - q en daartussen een lege opdracht. In de programmatekst mogen we puntkomma's toevoegen op plaatsen waar ze eigenlijk overbodig zijn, bijvoorbeeld aan het einde van een opdrachtenrij. Soms vereenvoudigt dit het samenstellen van de programmatekst en het onderhoud van het programma.

4.3 Conditionele structuren

Met de conditionele structuren wordt een opdrachtenrij slechts uitgevoerd indien aan een bepaalde conditie is voldaan. De conditie heeft de vorm van een logische uitdrukking. Bij de evaluatie van deze uitdrukking worden de waarden van de variabelen in de uitdrukking getest. De heersende toestand wordt evenwel niet gewijzigd. In Modula-2 onderscheiden we twee basisvormen : de conditiestructuur en de selectiestructuur.

De conditionele IF-opdracht

De syntaxis voor de conditionele IF-opdracht is :

IfOpdracht = 'IF' uitdrukking 'THEN' OpdrachtenRij 'END'

Het effect van de opdracht is :

- de waarde van de logische uitdrukking wordt berekend;
- als de waarde TRUE is wordt de opdrachtenrij tussen THEN en END uitgevoerd.

Voorbeelden :

Bereken het quotiënt van x en y, beide van het type INTEGER.

```
VAR x, y : INTEGER;

IF y # 0
THEN
  x := x DIV y
END
```

Gegeven zijn de INTEGER-variabelen a en b. Stel x gelijk aan het maximum van a en b :

```
VAR x, a, b : INTEGER;

x := a;
IF x < b
THEN
  x := b
END
(* x = maximum van a en b *)
```

Selectieve IF-opdracht

De syntaxis voor de selectieve IF-opdracht is

```
IfOpdracht = 'IF' uitdrukking 'THEN' OpdrachtenRij
                              'ELSE' OpdrachtenRij
             'END'
```

Effect van de opdracht :
- de waarde van de logische uitdrukking wordt berekend;
- als de waarde van de uitdrukking TRUE is wordt de opdrachtenrij na THEN uitgevoerd, anders wordt de opdrachtenrij na ELSE uitgevoerd.

Voorbeeld :

Gegeven zijn de INTEGER-variabelen a en b. Stel x gelijk aan het maximum van a en b.

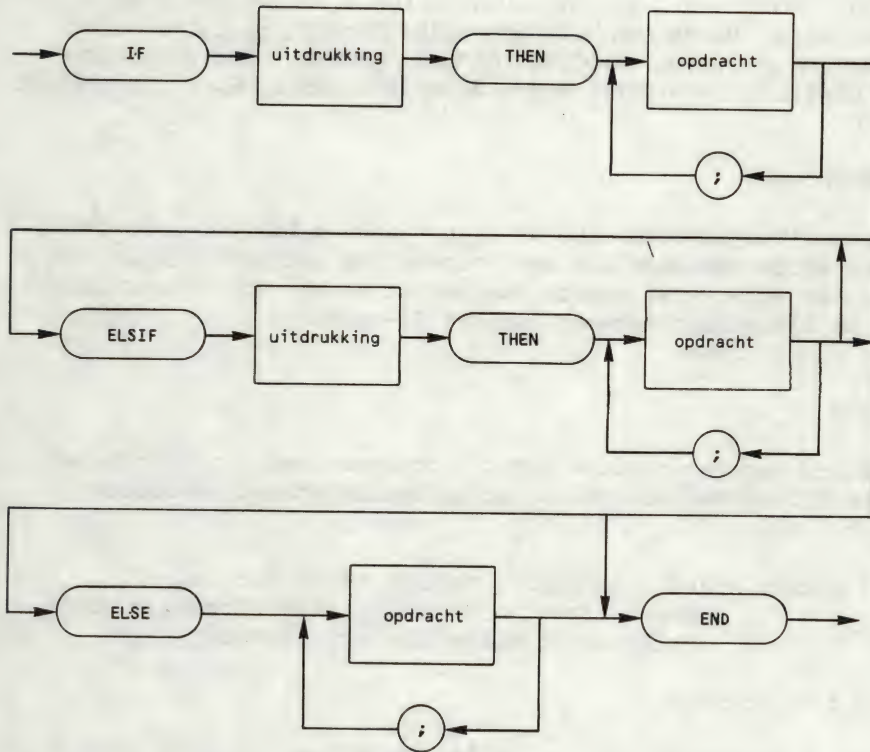
```
VAR x, a, b : INTEGER;

IF a > b
THEN
  x := a
ELSE
  x := b
END
(* x = maximum van a en b *)
```

Geneste selectieve IF-opdracht

Dikwijls is een selectie uit meer dan twee alternatieven noodzakelijk. Modula-2 beschikt hiervoor over een geneste IF-opdracht met de volgende syntaxis :

```
IfOpdracht = 'IF' uitdrukking 'THEN' OpdrachtenRij
             { 'ELSIF' uitdrukking 'THEN' OpdrachtenRij }
             [ 'ELSE' OpdrachtenRij ]
             'END'
```



De algemene vorm is

IF C_1
THEN
  S_1
ELSIF C_2
THEN
  S_2
ELSIF C_3
THEN
  S_3
...
ELSIF C_n
THEN
  S_n
ELSE
  S_{n+1}
END

De logische uitdrukkingen C_1, C_2, enzovoorts, worden na elkaar berekend. Zodra een logische uitdrukking C_i de waarde TRUE aflevert wordt de overeenkomstige opdrachtenrij S_i uitgevoerd. Alle verdere condities worden niet meer getest. Indien aan geen enkele conditie is voldaan wordt de opdrachtenrij S_{n+1} uitgevoerd.

De CASE-opdracht

Zeer dikwijls ontmoeten we situaties waarbij een opdrachtenrij moet worden gekozen uit een verzameling opdrachtenrijen aan de hand van de (discrete) waarde van een uitdrukking. Deze situatie kunnen we uitdrukken met een geneste IF-opdracht.

Voorbeeld :

De toestand van een bestand wordt aangegeven met een waarde van het type FileState (volgens de voorgestelde standaardmodule Files) :

```
TYPE FileState = (ok, nameError, noFile, existingFile,
                  deviceError, noMoreRoom, accessError, notOpen,
                  endError, outsideFile, otherError);

VAR res : FileState;
```

Bij elke fouttoestand van een bestand wordt een melding naar de standaarduitvoer geschreven :

```
nameError, noFile, existingFile :
   'fout tijdens het opzoeken van het bestand in de inhoudstabel'

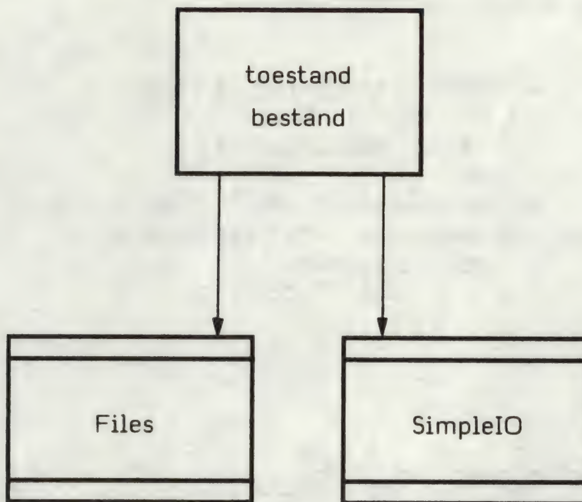
deviceError :
   'fout in de aandrijving'

noMoreRoom, accessError :
   'fout tijdens de toegang tot het bestand'

notOpen, endError, outsideFile :
   'foutieve bewerking op het bestand'

otherError :
   'niet gedefinieerde fout'
```

De toestand wordt bewaard in de variabele res. De teksten worden weergegeven met de bewerking WriteString van het abstracte datatype SimpleIO. Het abstractieschema voor dit programmafragment is dus :



figuur 4.1 De toestand van een bestand

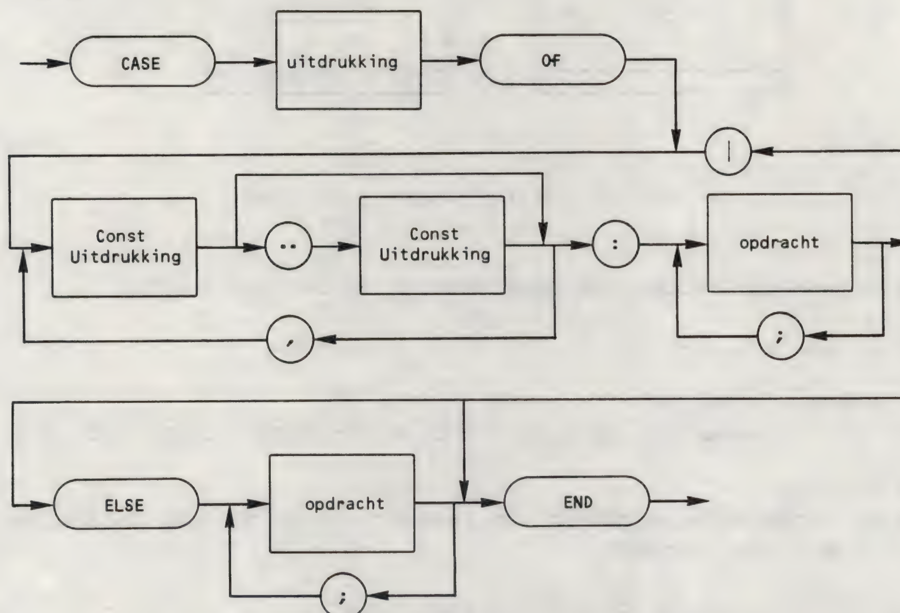
Met een geneste IF-opdracht programmeren we dat als volgt :

```
IF res = OK
THEN
  (* geen actie *)
ELSIF (res = nameError) OR (res = noFile) OR (res = existingFile)
THEN
  WriteString
  ('fout tijdens het opzoeken van het bestand in de inhoudstabel')
ELSIF (res = deviceError)
THEN
  WriteString('fout in de aandrijving')
ELSIF (res = noMoreRoom) OR (res = accessError)
THEN
  WriteString('fout tijdens de toegang tot het bestand')
ELSIF (res = notOpen) OR (res = endError) OR (res = outsideFile)
THEN
  WriteString('foutieve bewerking op het bestand')
ELSE
  WriteString('niet gedefinieerde fout')
END
```

Een geneste IF-opdracht, waarbij uitsluitend discrete waarden worden getest, is overzichtelijker als we bij het programmeren gebruik maken van een CASE-opdracht.

De syntaxis van de CASE-opdracht is :

```
CaseOpdracht      = 'CASE' uitdrukking OF case
                    { '|' case }
                    [ 'ELSE' OpdrachtenRij ]
                    'END'
case              = [ CaseWaardenLijst ':' OpdrachtenRij ]
CaseWaardenLijst = CaseWaarden [ ',' CaseWaarden ]
CaseWaarden       = ConstUitdrukking
                    [ '..' ConstUitdrukking ]
```



Het effect van de opdracht is :
- de waarde van de uitdrukking wordt berekend;
- indien de waarde voorkomt in een CaseWaardenLijst, wordt de bijhorende opdrachtenrij uitgevoerd. Indien de waarde niet in een waardenlijst voorkomt, wordt de opdrachtenrij na ELSE uitgevoerd.

De waarden in een CaseWaardenLijst zijn constant. Een waarde mag slechts in één enkele lijst voorkomen. Het type van alle waarden in de waardenlijsten moet overeenstemmen met het resultaattype van de uitdrukking.

Opmerkingen :
- De ELSE-clausule is facultatief. Indien deze clausule ontbreekt en indien ook de waarde van de uitdrukking in geen enkele lijst wordt gevonden, is het resultaat van de CASE-opdracht niet gedefinieerd.
- Volgens de syntaxis kan de inhoud van een 'case' leeg zijn. Dit betekent dat overbodige '|'-tekens kunnen worden toegevoegd. Het '|'-teken is een scheidingsteken voor de verschillende CASE-varianten naar analogie van de puntkomma voor opdrachten.

Voorbeelden :

We beschouwen opnieuw het voorbeeld voor het afdrukken van een boodschap afhankelijk van de toestand van een bestand.

```
CASE res OF
  ok :
  (* geen actie *)
| nameError..existingFile :
  WriteString
  ('fout tijdens het opzoeken van het bestand in de inhoudstabel')
| deviceError :
  WriteString('fout in de aandrijving')
| noMoreRoom, accessError :
  WriteString('fout tijdens de toegang tot het bestand')
| notOpen, endError, outsideFile :
  WriteString('foutieve bewerking op het bestand')
ELSE
  WriteString('niet gedefinieerde fout')
END
```

In het volgende voorbeeld steunt de selectie op de waarde van een teken :

```
VAR teken : CHAR;

CASE teken OF
  'Ø'..'9' :
  WriteString('cijfer')
| 'A'..'Z' :
  WriteString('hoofdletter')
| 'a'..'z' :
  WriteString('kleine letter')
| '*','-','+','/' :
  WriteString('operator')
ELSE
  WriteString('interpunctie of foutief teken')
END
```

Opmerking : Bij de vertaling van de CASE-opdracht worden de alternatieven in de gecompileerde code dikwijls geadresseerd via een tabel met sprongadressen. Het aantal elementen in deze tabel stemt overeen met het aantal alternatieven van de CASE-opdracht. Als dit aantal alternatieven te groot is, kan de compiler de CASE-opdracht niet correct verwerken. Bijvoorbeeld :

```
CASE getal OF
  0..9        : cijfers := 1
| 10..99      : cijfers := 2
| 100..999    : cijfers := 3
| 1000..9999  : cijfers := 4
ELSE            cijfers := 5
END
```

Dit voorbeeld kan veel beter worden gecodeerd met een geneste IF-opdracht. Maar ook als het aantal alternatieven kleiner is, bijvoorbeeld enkele tientallen, kunnen soms moeilijkheden voorkomen. De combinatie van een geneste IF-opdracht met CASE-opdrachten is dan een goed alternatief.

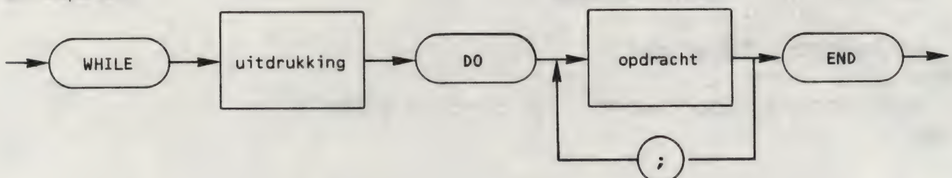
4.4 Herhalingsopdrachten

De kracht van een computer ligt in de eigenschap dat een aantal acties snel kan worden herhaald. Het herhaald uitvoeren is afhankelijk van vooraf berekende waarden van variabelen. In Modula-2 onderscheiden we de herhalingsopdrachten WHILE, FOR, REPEAT en LOOP.

De WHILE-opdracht

De WHILE-opdracht bestuurt het herhaald uitvoeren van een opdrachtenrij onder een bepaalde voorwaarde. De voorwaarde wordt weergegeven met een logische uitdrukking. De syntaxis voor de WHILE-opdracht is :

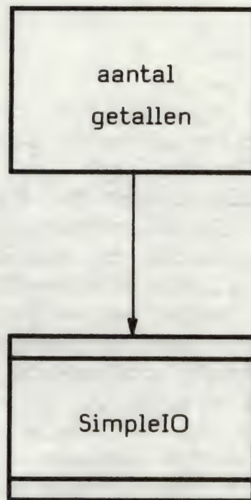
WhileOpdracht = 'WHILE' uitdrukking 'DO' OpdrachtenRij 'END'



Het effect van de opdracht is :

(1) de waarde van de logische uitdrukking wordt berekend;
(2) als deze waarde TRUE is wordt de opdrachtenrij tussen DO en END uitgevoerd. Daarna wordt (1) opnieuw uitgevoerd. Als de waarde FALSE is wordt de opdrachtenrij niet uitgevoerd.

Voorbeeld :
De invoer van een programma is een reeks positieve getallen groter dan nul. De reeks wordt afgesloten met het getal nul. We zoeken het aantal getallen in deze reeks. Voor de in- en uitvoer gebruiken we de standaardbestanden 'input' en 'output'. De bewerkingen op deze bestanden worden gegeven door het abstracte datatype SimpleIO. Hier hebben we slechts de bewerkingen 'ReadCard' en 'WriteCard' nodig om telkens een getal te lezen en het resultaat af te drukken. Het abstractieschema is :



figuur 4.2 Het aantal getallen in een reeks

```
VAR teller, x : CARDINAL;
    succes    : BOOLEAN;

BEGIN
teller := Ø;
(* teller = aantal getallen in de rij *)
```

```
ReadCard(x,succes);
WHILE x <> Ø DO
  (* teller = aantal getallen in de rij EN x # Ø *)
  INC(teller);
  ReadCard(x,succes)
END;

(* teller = aantal getallen in de rij EN x = Ø *)
WriteCard(teller,5)
END
```

Met de invoerstroom

11 24 52 Ø

wordt het resultaat 3 afgedrukt.

Met de invoerstroom

Ø

wordt aan de beginconditie voor de WHILE-opdracht niet voldaan. De opdrachtenrij tussen DO en END wordt daarom nooit uitgevoerd. Het programma geeft nul als resultaat.

De lus wordt gecontroleerd door de waarde van de variabelen die in de logische uitdrukking voorkomen. Aan deze variabelen moet dus vooraf een waarde worden toegekend. Is dit niet zo dan is het resultaat van de logische uitdrukking niet gedefinieerd. Tot slot moeten we ervoor zorgen dat de lus eindigt. Een noodzakelijke maar niet voldoende voorwaarde is dat de waarde van de variabele(n) in de logische uitdrukking tijdens de uitvoering van de opdrachtenrij wordt gewijzigd.

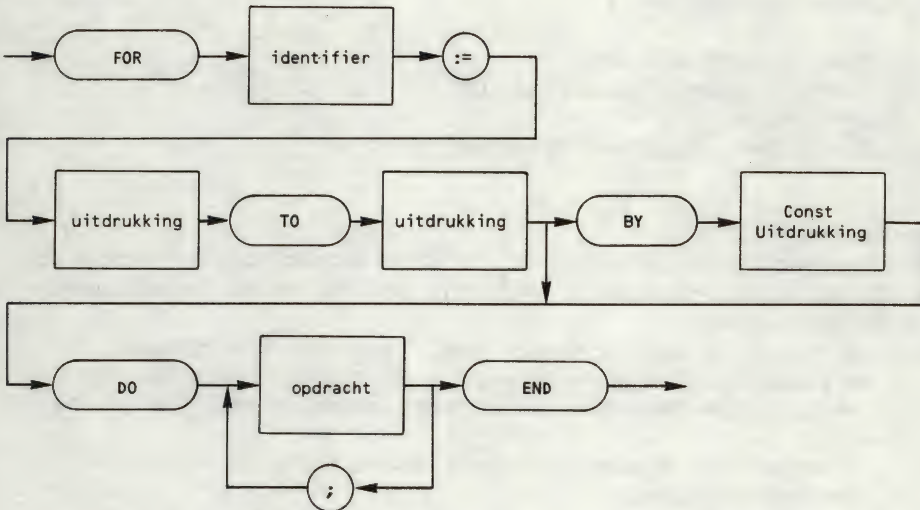
In het voorbeeld krijgt de variabele x een waarde door de leesopdracht vlak voor de WHILE-opdracht. In de opdrachtenrij kan de waarde van x ook met een leesopdracht worden gewijzigd.

De FOR-opdracht

De FOR-opdracht is een bijzondere vorm van een WHILE-opdracht : de lus wordt bestuurd door een variabele met een discreet type, bijvoorbeeld het type CHAR, INTEGER, CARDINAL of een enumeratietype. Na het uitvoeren van de opdrachtenrij wordt de waarde van de besturingsvariabele steeds met dezelfde constante waarde verhoogd of verlaagd. De lus eindigt indien de eindwaarde wordt overschreden.

De syntaxis voor de FOR-opdracht is :

```
ForOpdracht = 'FOR' identifier ':=' uitdrukking 'TO' uitdrukking
              [ 'BY' ConstUitdrukking ] 'DO'
              OpdrachtenRij
              'END'
```



De uitdrukkingen voor en na het 'TO'-symbool definiëren het bereik waartoe de waarde van de identifier moet behoren opdat de rij opdrachten wordt uitgevoerd. De uitdrukking na 'BY' geeft aan hoe de waarde van de besturingsvariabele wijzigt.

We onderscheiden twee mogelijkheden : de waarde van de constante uitdrukking is positief of negatief. In het eerste geval komt de FOR-opdracht

```
FOR identifier := uitdrukking1 TO uitdrukking2
   BY ConstUitdrukking DO
  OpdrachtenRij
END
```

in principe overeen met de opdrachtenrij :

```
identifier := uitdrukking1;
WHILE identifier <= uitdrukking2 DO
  OpdrachtenRij;
  INC(identifier, ConstUitdrukking)
END
```

Is de waarde van de constante uitdrukking negatief, dan komt de opdracht

```
FOR identifier := uitdrukking1 TO uitdrukking2
    BY ConstUitdrukking DO
  OpdrachtenRij
END
```

in principe overeen met :

```
identifier := uitdrukking1;
WHILE identifier >= uitdrukking2 DO
  OpdrachtenRij;
  DEC(identifier, ABS(ConstUitdrukking))
END
```

Opmerkingen :
- de BY-clausule is facultatief. Indien deze clausule ontbreekt, wordt bij elke iteratie de volgende waarde van de de besturingsvariabele genomen;
- de waarde van de besturingsvariabele is na de beëindiging van de iteratie niet gedefinieerd. De FOR-opdracht komt dus niet volledig overeen met de hiervoor vermelde WHILE-opdrachten :
  1. de waarde van de besturingsvariabele is na de iteratie niet gedefinieerd;
  2. de waarde van de besturingsvariabele mag in de opdrachtenrij niet worden gewijzigd.

De FOR-opdracht (zonder BY-clausule) wordt meestal in de volgende toepassing gebruikt :
- equidistante waarden van een gegeven interval moeten worden doorlopen;
- het aantal iteraties is vooraf bekend.

Voorbeelden :

De invoerstroom voor een programma is een reeks met 10 positieve getallen. We berekenen het gehele deel van het gemiddelde van deze getallen.

```
CONST maxAantal = 10;
TYPE  Teller    = [1..maxAantal];
VAR   som, x    : CARDINAL;
      i         : Teller;
      succes    : BOOLEAN;
```

```
BEGIN
som := Ø;

FOR i := 1 TO maxAantal DO
  ReadCard(x,succes);
  som := som + x
END;

WriteCard(som DIV maxAantal,5)
END
```

De variabele t van het type CHAR krijgt bij het uitvoeren van de volgende herhalingsstructuur de waarden

'A', 'F', 'K', 'P', 'U' en 'Z'

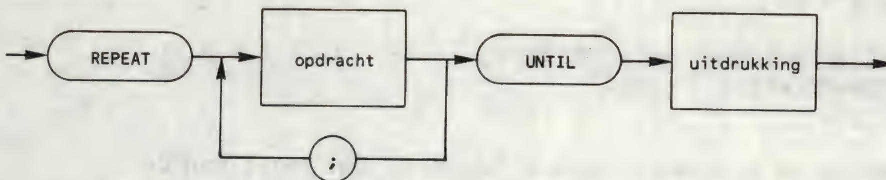
```
VAR t : CHAR;

FOR t := 'A' TO 'Z' BY 5 DO
  OpdrachtenRij
END
```

De REPEAT-opdracht

De REPEAT-opdracht bestuurt het herhaald uitvoeren van een opdrachtenrij tot aan een bepaalde voorwaarde is voldaan. De voorwaarde wordt weergegeven in een logische uitdrukking. De syntaxis voor de REPEAT-opdracht is :

RepeatOpdracht = 'REPEAT' OpdrachtenRij 'UNTIL' uitdrukking



Het effect van de opdracht is :
(1) de opdrachtenrij wordt uitgevoerd;
(2) de waarde van de logische uitdrukking wordt berekend. Indien deze waarde FALSE is wordt (1) opnieuw uitgevoerd. Als de waarde TRUE is stopt de lus.

Het essentiële verschil tussen de WHILE- en de REPEAT-opdracht is dat bij de WHILE-opdracht de logische uitdrukking voor het uitvoeren van de opdrachtenrij wordt berekend en bij de REPEAT-opdracht erna. De opdrachtenrij behorend tot een REPEAT-opdracht wordt dus steeds ten minste één keer uitgevoerd. We kunnen elke REPEAT-opdracht steeds tot een WHILE-opdracht omvormen (en omgekeerd) :

```
REPEAT OpdrachtenRij UNTIL uitdrukking
```

is equivalent met

```
OpdrachtenRij;
WHILE NOT uitdrukking DO
  (* uitdrukking = FALSE *)
  OpdrachtenRij
END
(* uitdrukking = TRUE *)
```

Voorbeelden :

De invoer van een programma is een reeks positieve getallen groter dan nul. De reeks wordt afgesloten met het getal nul. We zoeken het aantal getallen in deze reeks.

```
VAR teller, x : CARDINAL;
    succes    : BOOLEAN;
BEGIN
teller := Ø;
(* teller = aantal getallen in de reeks *)

REPEAT
  (* teller = aantal getallen in de reeks + 1 EN x # Ø *)
  INC(teller);
  ReadCard(x,succes)
UNTIL x = Ø;

(* teller = aantal getallen in de reeks + 1 EN x = Ø *)
WriteCard(teller - 1,5)
END
```

Het volgende programmafragment kopieert een regel van de standaardinvoer naar de standaarduitvoer. De datastructuren voor standaardin- en -uitvoer en de bewerkingen behoren tot het abstracte datatype SimpleIO. We gebruiken de bewerkingen :

- ReadChar : lees een teken van de standaardinvoer;
- WriteChar : schrijf een teken naar de standaarduitvoer.

Het einde van de regel testen we met de bewerking EOL (End Of Line).

```
VAR teken : CHAR;

REPEAT
  ReadChar(teken);
  WriteChar(teken)
UNTIL EOL()
```

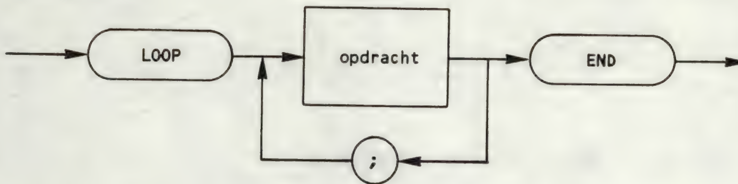
De LOOP-opdracht

De LOOP-opdracht bestuurt het herhaald uitvoeren van een opdrachtenrij. De LOOP-opdracht is een algemene vorm van de REPEAT- en de WHILE- opdracht : de opdracht voor de beëindiging van de lus kan op verschillende plaatsen in de opdrachtenrij voorkomen. De lus eindigt bij het uitvoeren van de EXIT-opdracht.

De syntaxis voor de LOOP-opdracht is :

```
LoopOpdracht = 'LOOP' OpdrachtenRij 'END'
ExitOpdracht = 'EXIT'
```

LoopOpdracht



ExitOpdracht



Een veel voorkomende vorm van deze structuur is :

```
LOOP
  OpdrachtenRij1;
  IF conditie THEN EXIT END;
  OpdrachtenRij2
END
```

De LOOP-opdracht eindigt meestal wanneer aan een bepaalde conditie is voldaan. Deze conditie testen we in een conditionele structuur.

Indien de $OpdrachtenRij_1$ leeg is komt de structuur :

```
LOOP
  IF conditie THEN EXIT END;
  OpdrachtenRij2
END
```

overeen met de WHILE-opdracht :

```
WHILE NOT conditie DO
  OpdrachtenRij2
END
```

Indien de $OpdrachtenRij_2$ leeg is komt de structuur :

```
LOOP
  OpdrachtenRij1;
  IF conditie THEN EXIT END
END
```

overeen met de REPEAT-opdracht :

```
REPEAT
  OpdrachtenRij1
UNTIL conditie
```

We beschrijven nog het effect van de algemene vorm :

```
LOOP
  OpdrachtenRij1;
  IF conditie THEN EXIT END;
  OpdrachtenRij2
END
```

(1) de $OpdrachtenRij_1$ wordt uitgevoerd;
(2) de logische uitdrukking wordt berekend. Indien de waarde TRUE is wordt de lus gestopt. In het andere geval wordt eerst $OpdrachtenRij_2$ en daarna (1) opnieuw uitgevoerd.

De LOOP-opdracht is geschikt voor toepassingen waarvan de logische structuur overeenstemt met deze algemene vorm. Voorbeelden hiervan zijn een (oneindig lang durend) interactief programma met de vorm :

```
LOOP
  lees opdracht;
  verwerk opdracht
END
```

en de verwerking van een invoerstroom tot een bepaalde waarde voorkomt :

```
LOOP
  lees (i-de waarde);
  test (i-de waarde);
  verwerk (i-de waarde)
END
```

Voorbeeld : een fragment van een interactief programma

De interactie van een computersysteem met de gebruiker verschilt van het ene systeem of zelfs van het ene programma tot het andere. In een menugestuurd systeem wordt een opsomming van mogelijke opdrachten op het scherm getoond. We kiezen dan de opdracht waarmee we een gegeven probleem kunnen oplossen. De vorm waarin we de opdracht geven verschilt van systeem tot systeem : soms moeten we de naam of het volgnummer intikken. In andere systemen kunnen we de gewenste opdracht aanwijzen met de cursor. De cursor verplaatsen we met bijzondere toetsen, de 'pijltjes', of met een speciaal toestel, 'een muis'. Op nog andere systemen zijn 'functietoetsen' op het toetsenbord beschikbaar.

In dit voorbeeld tonen we de logische structuur voor de interactie met een menu. We veronderstellen een toetsenbord met tien functietoetsen. De functies duiden we aan met de symbolen F1, F2, F3 en zo verder tot F1Ø. Voor de eerste zes functietoetsen is een opdracht gedefinieerd. We duiden deze acties aan met ActieEen, ActieTwee, ActieDrie, ActieVier, ActieVijf en ActieZes. De keuze van de overige functietoetsen heeft geen actie tot gevolg.

De opdrachtenrij van de lus bevat :
- SchrijfMenu : schrijft het menu op het scherm;
- WaardeFunctietoets : leest de invoer van de gebruiker en levert een waarde af die behoort tot een van de symbolen F1 tot F1Ø;
- keuze en uitvoering van de actie.

De functietoetsen definiëren we met een enumeratietype :

```
TYPE Functietoets = (F1,F2,F3,F4,F5,F6,F7,F8,F9,F1Ø)
```

De variabele

```
VAR functiekeuze : Functietoets
```

bevat de waarde waarmee de gewenste opdracht kan worden gekozen. De definitie in Modula-2 van de acties SchrijfMenu en WaardeFunctietoets wordt in het hoofdstuk 'Procedures en functies' beschreven. De structuur van het programmasegment is dan :

```
LOOP
  SchrijfMenu;
  functiekeuze := WaardeFunctieToets();
  CASE functiekeuze OF
    F1 : ActieEen
  | F2 : ActieTwee
  | F3 : ActieDrie
  | F4 : ActieVier
  | F5 : ActieVijf
  | F6 : ActieZes
  | F7..F10 : (* geen actie *)
  END
END
```

De CASE-opdracht bevat geen ELSE-clausule omdat alle mogelijke waarden voor de variabele functiekeuze in de variantlijsten worden vermeld. Eventueel kunnen we de variant

```
F7..F10 : (* geen actie *)
```

ook formuleren als

```
ELSE (* geen actie *)
```

Indien we de variabele functiekeuze verder niet in het programma gebruiken, is deze overbodig en herleiden we het fragment tot

```
LOOP
  SchrijfMenu;
  CASE WaardeFunctietoets() OF
    F1 : ActieEen
  | F2 : ActieTwee
  | F3 : ActieDrie
  | F4 : ActieVier
  | F5 : ActieVijf
  | F6 : ActieZes
  ELSE (* geen actie *)
  END
END
```

Voorbeeld : de verwerking van een gegevensstroom

Een programma leest een reeks getallen van het type CARDINAL tot het getal 9999 is gelezen. Na de invoer van 9999 wordt het rekenkundig gemiddelde van de getallen berekend, dit wil zeggen zonder dat rekening wordt gehouden met het afsluitgetal 9999. Een natuurlijke gedachtengang voor de opdrachten in de herhalingsstructuur om dit probleem op te lossen is :

(1) lees een getal;
(2) is het getal 9999 ? Zo ja, stop de iteratie;
(3) verwerk het getal en begin opnieuw bij (1).

Deze logische structuur wordt de lees-verwerk-strategie genoemd en stemt overeen met de algemene LOOP-opdracht.

Voor de oplossing gebruiken we de bewerkingen ReadCard en WriteCard, respectievelijk voor het lezen van en het schrijven naar de standaardin- en -uitvoer van een getal met type CARDINAL. Met de bewerking WriteString schrijven we een tekst naar de standaarduitvoer, de bewerking WriteLn begint een nieuwe regel.

```
CONST laatsteGetal = 9999;
VAR   teller       : CARDINAL;
      som          : CARDINAL;
      x            : CARDINAL;
      succes       : BOOLEAN;

BEGIN
teller := Ø;
som := Ø;

LOOP
  ReadCard(x,succes);
  IF x = laatsteGetal
  THEN
    EXIT
  END;
  som := som + x;
  teller := teller + 1
END;

IF teller > Ø
THEN
  WriteCard(som DIV teller,5);
  WriteLn
ELSE
  WriteString
    ('geen invoergetallen; gemiddelde niet gedefinieerd');
  WriteLn
END
END
```

We kunnen dit probleem ook oplossen met een WHILE-opdracht. De moeilijkheid is echter dat de lus wordt bestuurd door de nieuwe waarden die telkens worden gelezen. Eventueel wordt de opdrachtenrij van de WHILE-opdracht nooit uitgevoerd, namelijk als het eerste getal al gelijk is aan de afsluitwaarde. Het eerste getal wordt dus gelezen voor het begin van de lus.

```
lees getal;
WHILE test (i-de getal) DO
  verwerk (i-de getal);
  lees (i+1-de getal)
END
```

De structuur van de opdrachtenrij van de WHILE-opdracht wordt verwerk-lees-strategie genoemd. We geven nu de volledige oplossing :

```
CONST laatsteGetal = 9999;
VAR   teller       : CARDINAL;
      som          : CARDINAL;
      x            : CARDINAL;
      succes       : BOOLEAN;
BEGIN
teller := Ø;
som := Ø;

ReadCard(x,succes);
WHILE x <> laatsteGetal DO
  som := som + x;
  teller := teller + 1;
  ReadCard(x,succes)
END;

IF teller > Ø
THEN
  WriteCard(som DIV teller,5);
  WriteLn
ELSE
  WriteString
    ('geen invoergetallen; gemiddelde niet gedefinieerd');
  WriteLn
END

END
```

We kunnen een lees-verwerk-strategie ook implementeren met de WHILE- of de REPEAT-opdracht. Deze oplossingen vereisen wel een bijkomende variabele van het type BOOLEAN of een aanvullende conditie. Los dit probleem als oefening volgens de lees-verwerk-strategie op.

Opmerkingen :
- De LOOP-opdrachten kunnen we nesten. Een EXIT-opdracht geldt alleen voor de LOOP-opdracht waartoe EXIT behoort.
- De opdrachtenrij van een LOOP-opdracht kan meer dan één EXIT-opdracht bevatten. Beperk echter het gebruik van een LOOP-structuur met talrijke uitgangen!

- De EXIT-opdracht behoort steeds tot een LOOP-opdracht. Voor de andere herhalingsstructuren WHILE, REPEAT en FOR is geen gelijkwaardige opdracht beschikbaar.
- Bij voorkeur gebruiken we de WHILE- en de REPEAT-opdracht : bij deze structuren is de plaats van de eindconditie syntactisch duidelijk bepaald. Hierdoor verbeteren de leesbaarheid en de onderhoudbaarheid van het programma. Dit bezwaar vervalt meestal als de LOOP-opdracht slechts één enkele EXIT-opdracht bevat.

Samenvatting opdrachten

We kunnen nu de syntaxis voor een opdracht noteren als :

```
opdracht = [ ToekenningsOpdracht
           | WhileOpdracht
           | RepeatOpdracht
           | ForOpdracht
           | LoopOpdracht
           | IfOpdracht
           | CaseOpdracht
           | 'EXIT' ]
```

De opsomming is nog niet volledig. Nieuwe opdrachten worden in de volgende hoofdstukken toegevoegd.

4.5 De vorm van een programmamodule

Een Modula-2 programma is opgebouwd uit modulen. We onderscheiden de interne modulen, de programmamodulen, de definitie- en implementatiemodulen. De precieze betekenis van deze modulen wordt verklaard in het hoofdstuk 'Modulen'. Voorlopig nemen we aan dat een Modula-2 programma alleen bestaat uit een programmamodule. De syntaxis voor een dergelijke module is :

```
ProgrammaModule = MODULE identifier ';'
                  { import }
                  blok
                  identifier '.'
```

De kop van een module bestaat uit het symbool 'MODULE' gevolgd door de naam van de module, bijvoorbeeld :

```
MODULE Gemiddelde;
```

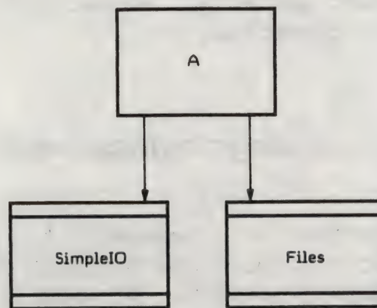
Daarna volgen eventueel een of meer IMPORT-lijsten. Een IMPORT-lijst refereert aan de objecten die in andere modulen worden gedefinieerd. Sommige van deze modulen kunnen we beschouwen als abstracte datatypen : in deze modulen worden gegevensstructuren en de bewerkingen hierop gedefinieerd. Een voorbeeld van zo'n abstract datatype is SimpleIO. In deze module worden de gegevensstructuren 'input' en 'output' voor de standaardin- en -uitvoer gedefinieerd. Voorlopig gebruiken we de volgende syntaxis voor de IMPORT-lijst :

```
import = 'FROM' identifier 'IMPORT' IdentLijst ';'
```

De identifier verwijst naar de module waaruit we objecten willen gebruiken. De IdentLijst is de opsomming van de objecten uit de desbetreffende module.

Voorbeeld :

In module A gebruiken we objecten 'ReadChar' en 'WriteChar' uit de module 'SimpleIO' en de objecten 'File' en 'Create' uit de module 'Files'. We stellen dit voor met het abstractieschema



figuur 4.3 A verwijst naar SimpleIO en Files

en in de tekst met

```
FROM SimpleIO IMPORT ReadChar, WriteChar;
FROM Files    IMPORT File, Create;
```

Na de IMPORT-lijsten wordt het blok gedefinieerd. De voorlopige definitie van de syntaxis voor een blok is :

```
blok = { declaraties }
       'BEGIN'
       OpdrachtenRij
       'END'
```

In het eerste deel van het blok worden alle objecten gedefinieerd die we in de programmamodule zullen gebruiken. In een programmamodule volgt na het BEGIN-symbool de opdrachtenrij. De opdrachtenrij eindigt met het END-symbool. Tot slot eindigt de module met de herhaling van de moduleidentifier gevolgd door de punt.

Voorbeeld :
We programmeren een volledige programmamodule voor het volgende probleem :

Een programma leest een reeks getallen met type CARDINAL tot het getal 9999 is gelezen. Na de invoer van 9999 wordt het rekenkundig gemiddelde van de getallen berekend, dit wil zeggen zonder dat rekening wordt gehouden met het afsluitgetal 9999.

```
MODULE Gemiddelde;
FROM SimpleIO IMPORT ReadCard,
                     WriteCard, WriteLn, WriteString;

CONST laatsteGetal = 9999;
VAR   teller       : CARDINAL;
      som          : CARDINAL;
      x            : CARDINAL;
      succes       : BOOLEAN;

BEGIN
teller := Ø;
som := Ø;

LOOP
  ReadCard(x,succes);
  IF x = laatsteGetal
  THEN
    EXIT
  END;
  som := som + x;
  teller := teller + 1
END;
```

```
IF teller > Ø
THEN
  WriteCard(som DIV teller,5);
  WriteLn
ELSE
  WriteString
    ('geen invoergetallen; gemiddelde niet gedefinieerd');
  WriteLn
END

END Gemiddelde.
```

Literatuur :

Buhr P. A., 'A Case for Teaching Multi-exit Loops to Beginnning Programmers',
Sigplan, november 1985

Johnson M. C., 'Pascal's Design Flaws, Case Limitations',
Byte, maart 1984

Modula-2 Standard Library Definition Modules,
Modula-2 News #1, januari 1985

Muller H. A., 'Differences Between Modula-2 and Pascal',
Sigplan, oktober 1984

Van Amstel J. J., 'Programmeren : het ontwerpen van algoritmen met PASCAL', Academic Service, Den Haag 1984

Soloway E., Bonar J. en Ehrlich K., 'Cognitive Strategies and Looping Constructs', Communications of ACM #11, november 1983

Wirth N., 'Programming in Modula-2', derde verbeterde editie,
Springer Verlag, Berlijn 1985

Wirth N., 'Revisions and Amendements to Modula-2',
ETH, Zurich 1984

Oefeningen

1. Herschrijf de volgende code met ELSIF :

```
IF P1
THEN
  IF P2
  THEN
    S1
  ELSE
    S2
  END
ELSE
  IF P3
  THEN
    S3
  END
END
```

2. Wat is de uitvoer voor elk van de volgende programma-fragmenten ?

a.
```
n := 3;
FOR i := 4 TO n DO
  WriteCard(i, 3)
END
```

b.
```
i := 3;
j := 7;
WHILE NOT ((i < 3) OR (j < 7)) DO
  INC(i);
  DEC(j);
  WriteCard(i, 3);
  WriteCard(j, 3)
END
```

c.
```
n := 3;
i := 4;
REPEAT
  WriteCard(i, 3);
  INC(i)
UNTIL i > n
```

```
d. i := 3;
   j := 7;
   LOOP
     DEC(j);
     IF (i < 3) OR (j < 3)
     THEN
       EXIT
     END;
     INC(i);
     WriteCard(i, 3);
     WriteCard(j, 3)
   END
```

3. Bereken het gemiddelde van een reeks getallen. De reeks wordt afgesloten met de waarde 9999. Gebruik de lees-verwerk-strategie en de WHILE-opdracht.

4. Zoals opgave 3; gebruik de REPEAT-opdracht.

5. Vorm de volgende WHILE-opdracht om tot een LOOP-opdracht :

```
LOOP
  ReadCard(i, s);
  IF (i < 1) OR (i > 5)
  THEN
    EXIT
  END;
  WriteCard(i, 3)
END
```

6. De invoer voor een programma is een reeks getallen. De reeks wordt afgesloten met het getal 9999. Dit getal behoort niet tot de eigenlijke invoer. Druk de reeks getallen geroteerd af : het tweede getal wordt eerst afgedrukt, het derde getal wordt als tweede afgedrukt en zo verder. Tot slot wordt het eerste gelezen getal afgedrukt. Test het programma met de volgende gegevens :

a. Ø 1 2 3 4 5 6 7 8 9 9999
b. 1 9999
c. 9999

7. De Fibonacci-getallen worden gedefinieerd als

$fib_0 = 0$

$fib_1 = 1$

$fib_n = fib_{n-1} + fib_{n-2}$

Schrijf een programmamodule voor het berekenen van de eerste tien Fibonacci-getallen.

8. De invoer voor een programmamodule is twee getallen m en n van type CARDINAL, waarbij geldt m > n. Schrijf een programmamodule voor het berekenen van alle Fibonacci-getallen begrepen tussen m en n.

9. De invoer voor een programmamodule is het getal m van type CARDINAL. Bereken het eerste Fibonacci-getal groter dan m.

5 Samengestelde typen

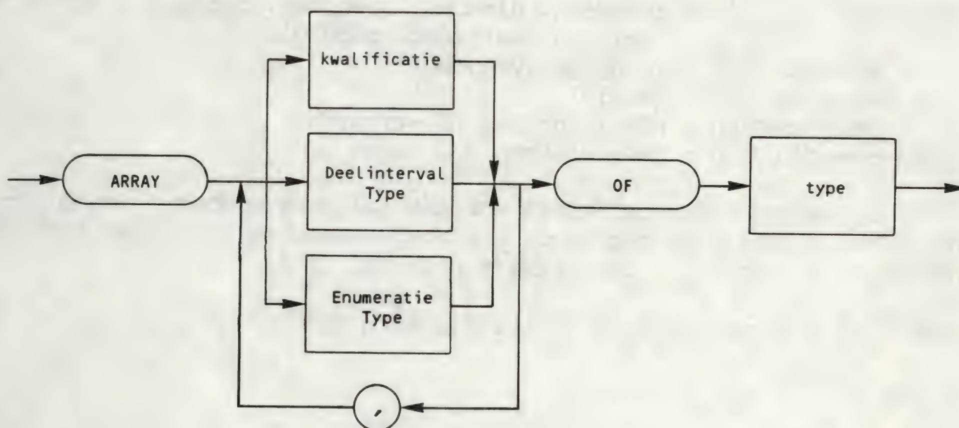
De tot nu toe besproken typen variabelen kunnen slechts enkelvoudige waarden aannemen. Sommige grootheden worden echter gekarakteriseerd door meer dan één waarde. Deze grootheden hebben een samengestelde waarde die uit een aantal componenten bestaat. Iedere component heeft een enkelvoudige waarde of is zelf weer een samengestelde waarde.

In Modula-2 onderscheiden we de volgende samengestelde typen : het tabeltype ARRAY, het recordtype RECORD, het verzamelingtype SET en het wijzertype POINTER. Het POINTER-type wordt in een volgend hoofdstuk beschreven.

5.1 Het tabeltype ARRAY

Een ARRAY-waarde, een tabel, bestaat uit een aantal componenten van hetzelfde type, het componenttype. De componenten, of de elementen van een tabel, duiden we aan met een index. Ook de index heeft weer een bepaald type, het indextype. Het basistype van de index is steeds een discreet type : een deelintervaltype, een enumeratietype, het type CHAR, BOOLEAN, CARDINAL of INTEGER. We declareren een tabel volgens de volgende syntaxis :

TabelType



```
TabelType        = 'ARRAY' EnkelvoudigType 'OF' type
EnkelvoudigType = EnumeratieType | DeelintervalType
                  | kwalificatie
```

Een tabel 'a' met 100 elementen van het type CARDINAL en waarvan de indices begrepen zijn tussen 1 en 100, declareren we als :

```
VAR a : ARRAY [1..100] OF CARDINAL;
```

Het type van deze tabel is onbenoemd. Het is beter de tabel als volgt te definiëren

```
CONST bovengrens = 100;
TYPE  Index      = [1..bovengrens];
      Tabel      = ARRAY Index OF CARDINAL;
VAR   a          : Tabel;
```

De ondergrens hoeft niet gelijk aan één te zijn. Elke waarde is toegelaten, op voorwaarde dat steeds geldt

ondergrens <= bovengrens

```
CONST ondergrens = -10;
      bovengrens = 20;
TYPE  Index      = [ondergrens..bovengrens];
      Tabel      = ARRAY Index OF CARDINAL;
VAR   b          : Tabel;
```

De variabele b is een tabel met 31 elementen : b_{-10}, b_{-9}, en zo verder tot b_{19} en b_{20}. Het basistype voor Index is INTEGER.

In het volgende voorbeeld definiëren we een tabel waarin per werkdag het aantal voor het bedrijf IT gewerkte uren wordt geregistreerd. In dit vooruitstrevende bedrijf kan het aantal werkuren per dag door de werknemer zelf worden bepaald (tussen zes en tien uur per dag).

```
TYPE Dag          = (maandag, dinsdag, woensdag, donderdag,
                     vrijdag, zaterdag, zondag);
     Werkdag      = [maandag..vrijdag];
     Werkuren     = [6..10];
     GewerkteUren = ARRAY Werkdag OF Werkuren;
VAR  gewerkteUren : GewerkteUren;
```

Elk element van een tabel wordt geselecteerd door de naam van de tabelvariabele gevolgd door een waarde van het indextype tussen de haken '[' en ']'. De syntaxis hiervoor is :

```
aanwijzing = kwalificatie '[' uitdrukking ']'
```

Voorbeelden :

Het element b_{-2} krijgt de waarde 120 met de opdracht

```
b[-2] := 120;
```

Het aantal gewerkte uren op dinsdag registreren we met de opdracht

```
gewerkteUren[dinsdag] := 7;
```

We berekenen het aantal werkuren per week voor een werknemer :

```
VAR totaalWerkuren   : CARDINAL;
    dag              : Werkdag;
    gewerkteUren     : ARRAY Werkdag OF Werkuren;

totaalWerkuren := 0;
FOR dag := maandag TO vrijdag DO
  totaalWerkuren := totaalWerkuren + gewerkteUren[dag]
END
```

De standaardfunctie HIGH levert de waarde op van de bovengrens van de index van een tabel. Het type van het resultaat stemt overeen met het indextype.

```
VAR a : ARRAY [1..100] OF CHAR;
    b : ARRAY ['A'..'E'] OF CARDINAL
```

De functie-aanroep HIGH(a) levert de waarde 100 op, de aanroep HIGH(b) de waarde 'E'. Met deze functie kunnen we zeer algemene programma's schrijven voor bewerkingen op tabellen. We illustreren dit in de volgende voorbeelden :

We definiëren het tabeltype Tabel met de declaratie

```
CONST n = 100;
TYPE  Index  = [1..n];
      Tabel  = ARRAY Index OF CARDINAL;
      Plaats = [0..n];
```

Gegeven zijn de variabelen

```
VAR   b      : Tabel;
      x, min : CARDINAL;
      p      : Plaats;
```

Eerst zoeken we de plaats van de variabele x in tabel b. Als x = b_i krijgt p de waarde i. Als x niet in de tabel voorkomt krijgt p de waarde 0. We onderzoeken de tabel met de lineaire zoekmethode :

met de variabele i selecteren we achtereenvolgens de elementen van b en we vergelijken deze met x. De lus stopt als $x = b_i$ of als $i = n$.

```
VAR i : Index;

i := 1;
LOOP
  (* als x in b voorkomt dan komt x voor in bi tot bn *)
  IF b[i] = x
  THEN
    p := i;
    EXIT
  END;
  IF i = HIGH(b)
  THEN
    p := 0;
    EXIT
  END;
  INC(i)
END
```

Nu zoeken we het kleinste element van de tabel :

```
min := b[1];
FOR i := 2 TO HIGH(b) DO
  (* min is het minimum van b1 tot bi *)
  IF b[i] < min
  THEN
    min := b[i]
  END
END
```

In het volgende voorbeeld sorteren we tabel b van klein naar groot zodat geldt :

$$b_i <= b_j \text{ voor } i < j$$

We gebruiken de tussenvoegmethode. De tussenvoegmethode wordt ook gebruikt in het kaartspel : elke speler houdt bij het delen de kaarten geordend in de hand. Wanneer nieuwe kaarten worden gegeven, steekt hij deze er op de juiste plaats tussen. Als we de tabel b[1..n] sorteren, beginnen we met de deeltabel b[1..1]. Daarna voegen we het element b[2] toe op de juiste plaats. We herhalen deze werkwijze tot b[n].

```
CONST n      = 100;
TYPE  Index  = [1..n];
      Tabel  = ARRAY Index OF CARDINAL;
      Plaats = [0..n];
```

```
VAR i      : Index;
    h      : CARDINAL;
    j      : Plaats;
    b      : Tabel;

FOR i := 2 TO HIGH(b) DO
  (* de elementen b1 tot bi zijn geordend *)
  j := i;
  WHILE (j > 1) AND (b[j - 1] > b[j]) DO
    h := b[j];
    b[j] := b[j - 1];
    b[j - 1] := h;
    DEC(j)
  END
END
```

We kunnen dit fragment nog optimaliseren : bij elke iteratie van de FOR-opdracht wordt aan de variabele h de waarde b_i toegekend. De opdrachten waarin h voorkomt, verwijderen we uit de opdrachtenrij van de WHILE-opdracht :

```
FOR i := 2 TO HIGH(b) DO
  (* de elementen b1 tot bi zijn geordend *)
  j := i;
  h := b[i];
  WHILE (j > 1) AND (b[j - 1] > h) DO
    b[j] := b[j - 1];
    DEC(j)
  END;
  b[j] := h
END
```

Met deze code verschuiven we de elementen van b naar de plaats die we met de opdracht

h := b[i]

hebben vrijgemaakt. De waarde h plaatsen we op de vrije plaats die is ontstaan na de afloop van de schuifopdrachten.

Als laatste voorbeeld zoeken we de positie van de variabele x in de geordende tabel b. Het antwoord registreren we weer in de variabele p : p = Ø als x niet in de tabel voorkomt; anders gelden 1 <= p <= n en b[p] = x.

Met de binaire zoekmethode wordt steeds rekening gehouden met een deel van de tabel waarin x kan voorkomen. In het begin beschouwen we de volledige tabel. De deeltabel verkleinen we door telkens het middelste element te vergelijken met x. Daarna wordt de onderste of de bovenste helft niet meer bekeken. De lus stopt wanneer we x hebben gevonden of wanneer de deeltabel leeg is.

De grenzen van de deeltabel die we willen onderzoeken, geven we aan met de variabelen ondergrens en bovengrens. De ondergrens en de bovengrens worden geïnitieerd op 1 en HIGH(b). De deeltabel is leeg indien

ondergrens > bovengrens

Aan deze voorwaarde is voldaan als x niet in de tabel voorkomt. De variabele p krijgt de waarde nul. Het middelste element wijzen we aan met de variabele midden.

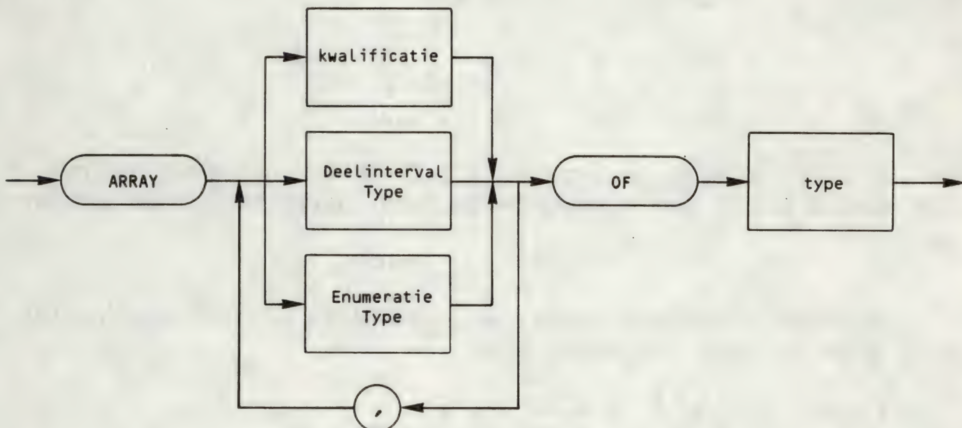
```
CONST n          = 1ØØ;
TYPE  Index      = [1..n];
      Tabel      = ARRAY Index OF CARDINAL;
      Grens      = [Ø..n + 1];
      Plaats     = [Ø..n];
VAR   ondergrens : Grens;
      bovengrens : Grens;
      midden     : Index;
      p          : Plaats;
      b          : Tabel;

ondergrens := 1;
bovengrens := HIGH(b);
LOOP
  (* als x voorkomt in b dan behoort x tot
     b[ondergrens..bovengrens] *)
  IF ondergrens > bovengrens
  THEN
    p := Ø;
    EXIT
  END;
  midden := (ondergrens + bovengrens) DIV 2;
  IF b[midden] < x
  THEN
    ondergrens := midden + 1
  ELSIF b[midden] = x
  THEN
    p := midden;
    EXIT
  ELSE
    bovengrens := midden - 1
  END
END
```

Meerdimensionale tabellen

Alle elementen van een tabel hebben hetzelfde type. Elk element kan op zich weer worden gekarakteriseerd door een tabel. Een matrix of een meerdimensionale tabel is een tabel van tabelelementen. De algemene syntaxis voor het tabeltype wordt gegeven door :

```
TabelType        = 'ARRAY' EnkelvoudigType
                           { ',' EnkelvoudigType } 'OF' type
EnkelvoudigType = DeelintervalType | EnumeratieType | kwalificatie
```



Voorbeelden :
Een matrix kunnen we beschouwen als een tabel (rij) waarvan elk element weer een tabel (kolom) is. De algemene declaratie voor een matrix is :

```
ARRAY EnkelvoudigType OF
  ARRAY EnkelvoudigType OF type
```

De declaratie voor tabel a met drie rijen en vier kolommen is dan

```
VAR a : ARRAY [1..3] OF ARRAY [1..4] OF type
```

De syntaxis laat ook de volgende verkorte notatie toe :

```
VAR a : ARRAY [1..3], [1..4] OF type
```

Een schaakbord definiëren we als volgt :

```
TYPE Stuk = (leeg, pion, loper, paard, toren, koning,
             koningin);
     Schaakbord = ARRAY [1..8],['a'..'h'] OF Stuk;
VAR  s : Schaakbord;
```

De elementen van een tabel worden geselecteerd volgens de syntaxis :

```
selectie  = kwalificatie{ '[' UitdLijst ']' }
UitdLijst = uitdrukking{ ',' uitdrukking }
```

Voorbeeld :
De matrix a ziet er als volgt uit :

1	2	3	4
5	6	7	8
9	10	11	12

De waarde van a[1,2] is 2 en van a[3,1] is 9. Als door een zet op het veld s[1,'c'] een paard terecht komt, geven we dit aan met de opdracht

```
s[1,'c'] := paard
```

Ook voor de meerdimensionale tabellen is de standaardfunctie HIGH gedefinieerd. De declaratie

```
TYPE Matrix = ARRAY A, B, C OF type;
VAR  tabel  : Matrix;
```

is gelijkwaardig met

```
TYPE Matrix = ARRAY A OF
                ARRAY B OF
                  ARRAY C OF type
```

De functie-aanroep HIGH(tabel) levert de bovengrens van het type A op, de aanroep HIGH(tabel[i]), waarbij i van het type A is, levert de bovengrens van B op en tot slot levert de aanroep HIGH(tabel[i,j]), waarbij i van het type A is en j van het type B, de bovengrens van C.

Voorbeeld :
Gegeven is de declaratie

```
TYPE Tabel = ARRAY [1..5],[1..3],['a'..'p'] OF CARDINAL;
VAR  a     : Tabel;
```

De functie-aanroepen van HIGH leveren de volgende resultaten op :

```
HIGH(a)       ->     5
HIGH(a[1])    ->     3
HIGH(a[2,3])  ->    'p'
```

De toekenningsopdracht

De toekenning is gedefinieerd voor elementen van het tabeltype. De toekenningsopdracht kan dus zowel worden uitgevoerd tussen variabelen van het tabeltype als tussen de elementen waaruit ze worden samengesteld.

Gegeven zijn de matrices a en b :

```
TYPE Tabel = ARRAY [1..5],[1..3] OF CARDINAL;
VAR  a, b  : Tabel;
```

De opdracht

```
a := b
```

is gelijkwaardig met de opdrachtenrij

```
FOR i := 1 TO HIGH(a) DO
  FOR j := 1 TO HIGH(a[1]) DO
    a[i,j] := b[i,j]
  END
END
```

We definiëren

```
VAR a : ARRAY [1..3],[1..3] OF CARDINAL;
    b : ARRAY [1..3],[1..3] OF CARDINAL;
```

Voor deze definitie is de toekenningsopdracht, bijvoorbeeld a := b, niet toegelaten; a en b hebben een verschillend onbenoemd type. Verschillende onbenoemde typen zijn niet overdraagbaar, zelfs indien ze dezelfde gegevensstructuren voorstellen. In de plaats van a := b zullen we nu moeten schrijven :

```
FOR i := 1 TO HIGH(a) DO
  FOR j := 1 TO HIGH(a[1]) DO
    a[i,j] := b[i,j]
  END
END
```

Als we de i-de en j-de rij van matrix a willen omwisselen, schrijven we :

```
TYPE Tabel   = ARRAY [1..m],[1..n] OF CARDINAL;
VAR  a       : Tabel;
     h       : CARDINAL;
     i, j, k : CARDINAL;

FOR k := 1 TO HIGH(a[1]) DO
  h := a[i,k];
  a[i,k] := a[j,k];
  a[j,k] := h
END
```

Met de declaratie

```
TYPE Rij   = ARRAY [1..n] OF CARDINAL;
     Tabel = ARRAY [1..m] OF Rij;
VAR  a     : Tabel;
     h     : Rij;
     i, j  : CARDINAL;
```

wordt elke rij van de tabel als een afzonderlijk object beschouwd. Twee rijen kunnen we nu omwisselen met de opdrachten :

```
h := a[i];
a[i] := a[j];
a[j] := h
```

Bekijk nu de declaraties

```
TYPE Tabel = ARRAY [1..5],[1..3] OF CARDINAL;
VAR  a     : Tabel;
```

Door de opdrachtenrij

```
k := 1;
FOR i := 1 TO HIGH(a) DO
  FOR j := 1 TO HIGH(a[1]) DO
    a[i,j] := k;
    INC(k)
  END
END
```

ziet a er als volgt uit :

```
 1     2     3
 4     5     6
 7     8     9
10    11    12
13    14    15
```

Het produkt van twee matrices

Behalve de toekenningsopdracht zijn er geen andere bewerkingen met tabellen toegelaten. Deze bewerkingen, bijvoorbeeld de som van twee matrices, programmeren we steeds element per element. We illustreren dit met een volledige programmamodule waarbij twee matrices worden gelezen. Daarna berekenen we het uitwendig produkt van de matrices en drukken we de resultaatmatrix af.

```
MODULE Produkt;
FROM SimpleIO IMPORT ReadCard,
                     WriteCard, WriteLn, WriteString, WriteChar;

TYPE T1      = ARRAY [1..3],[1..4] OF CARDINAL;
     T2      = ARRAY [1..4],[1..2] OF CARDINAL;
     T3      = ARRAY [1..3],[1..2] OF CARDINAL;
VAR  a       : T1;
     b       : T2;
     c       : T3;
     i, j, k : CARDINAL;
     succes  : BOOLEAN;

BEGIN
(* lees tabel a *)
WriteString('tabel a :');
WriteLn;
FOR i := 1 TO HIGH(a) DO
  FOR j := 1 TO HIGH(a[1]) DO
    ReadCard(a[i,j], succes);
    WriteChar(' ');
  END;
  WriteLn
END;

(* lees tabel b *)
WriteString('tabel b :');
WriteLn;
```

```
FOR i := 1 TO HIGH(b) DO
  FOR j := 1 TO HIGH(b[1]) DO
    ReadCard(b[i,j], succes);
    WriteChar(' ');
  END;
  WriteLn
END;

(* bereken produkt c = a * b *)
FOR i := 1 TO HIGH(a) DO
  FOR j := 1 TO HIGH(b[1]) DO
    c[i,j] := 0;
    FOR k := 1 TO HIGH(a[1]) DO
      c[i,j] := c[i,j] + a[i,k] * b[k,j]
    END
  END
END;

(* druk c af *)
FOR i := 1 TO HIGH(c) DO
  FOR j := 1 TO HIGH(c[1]) DO
    WriteCard(c[i,j],5)
  END;
  WriteLn
END

END Produkt.
```

Voorbeeld :

De invoer voor tabel a is :

```
1 2 3 1
1 0 2 1
3 2 1 1
```

en voor tabel b :

```
1 2
1 1
1 1
1 1
```

De afgedrukte resultaatmatrix is :

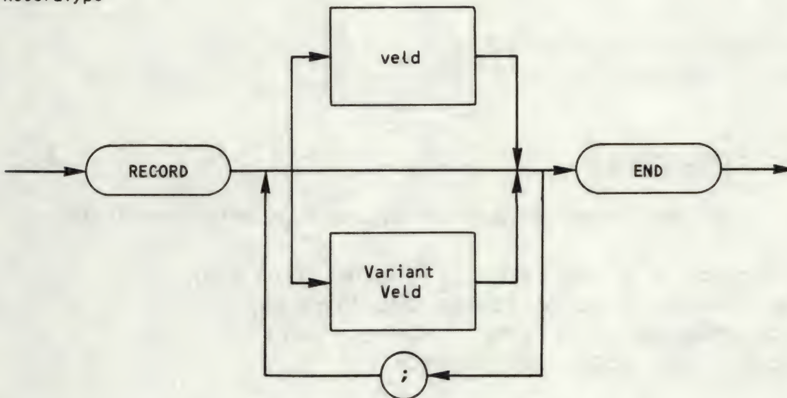
```
7    8
4    5
7   10
```

5.2 Het recordtype RECORD

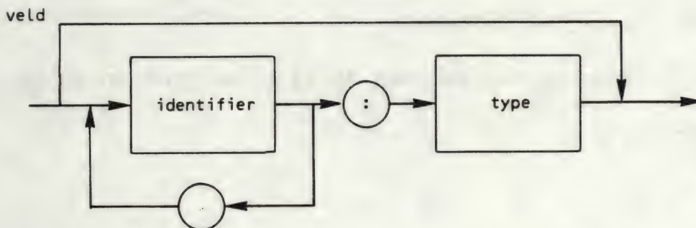
Alle elementen van een tabel behoren tot hetzelfde type. Met een record kunnen we nieuwe typen definiëren waarvan de waarde is samengesteld uit componenten met een verschillend type. Een datum kan bijvoorbeeld samengesteld zijn uit de componenten jaar, maand en dag. Een persoonsbeschrijving kan bestaan uit een naam, een geboortedatum, een adres en het geslacht. Sommige van deze componenten kunnen opnieuw een recordstructuur vormen : bijvoorbeeld geboortedatum, adres. De componenten van een recordstructuur worden velden genoemd. De naam van een veld is de veldidentificatie. We definiëren een recordstructuur met de volgende syntaxis :

```
RecordType  = 'RECORD' VeldenLijst 'END'
VeldenLijst = veld{ ';' veld }
veld        = [ IdentLijst ':' type ]
```

RecordType



veld



Voorbeelden :

```
TYPE Maand = (januari, februari, maart, april, mei, juni, juli,
              augustus, september, oktober, november, december);
     Datum = RECORD
       dag   : [1..31];
       maand : Maand;
       jaar  : [1900..2000]
     END;
     Persoon = RECORD
       voornaam, familienaam : ARRAY [0..20] OF CHAR;
       geslacht : (man,vrouw);
       geboortedatum : Datum;
       adres = RECORD
         straat     : ARRAY [0..40] OF CHAR;
         woonplaats : ARRAY [0..25] OF CHAR
       END
     END;
```

De velden van een record worden geselecteerd met de veld-identificatie volgens de syntaxis :

```
aanwijzing = kwalificatie{ selectie }
selectie   = '.' identifier | '[' UitdLijst ']'
```

Voorbeeld :
VAR deelnemer : Persoon;

De velden van een deelnemer worden onder meer geselecteerd met

deelnemer.voornaam[i] : het teken$_{i+1}$ van de voornaam;
deelnemer.familienaam : de volledige familienaam;
deelnemer.geboortedatum.jaar : het geboortejaar;
deelnemer.adres : het volledige adres.

Natuurlijk kunnen we ook tabellen definiëren waarvan de elementen records zijn :

VAR student : ARRAY [1..23] OF Persoon;

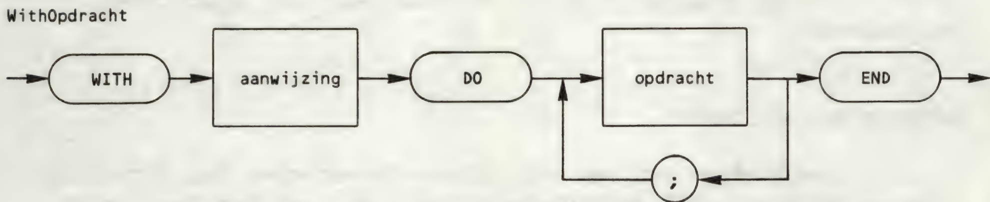
De familienaam van de tiende student van de klas selecteren we met

student[10].familienaam

De WITH-opdracht

Naar analogie van de FOR-opdracht voor tabellen, is voor records de WITH-opdracht gedefinieerd. Met deze opdracht verwijzen we naar de velden van een record zonder dat we de recordnaam telkens hoeven te herhalen. De syntaxis voor de WITH-opdracht is :

WithOpdracht = 'WITH' aanwijzing 'DO' OpdrachtenRij 'END'



Met de WITH-opdracht wordt een bepaald bereik gedefinieerd. Binnen dit bereik worden de veldnamen als variabelen gebruikt.

Voorbeeld :

```
WITH deelnemer DO
  voornaam := 'Eric';
  familienaam := 'Verhulst';
  geslacht := man;
  WITH geboortedatum DO
    dag := 11;
    maand := augustus;
    jaar := 1949
  END
END
```

De aanwijzing in de WITH-opdracht kan ook een element zijn van een tabel met recordvelden :

```
WITH student[i] DO
  OpdrachtenRij
END
```

De waarde van de variabele in de WITH-opdracht mag niet in de opdrachtenrij worden gewijzigd. Deze beperking geldt ook bij de FOR-opdracht waar de besturingsvariabele ook niet in de opdrachtenrij mag worden gewijzigd.

We vullen nu de syntaxis voor opdrachten met de WITH-opdracht aan :

```
opdracht = [ ToekenningsOpdracht
           | WhileOpdracht
           | RepeatOpdracht
           | ForOpdracht
           | LoopOpdracht
           | IfOpdracht
           | CaseOpdracht
           | WithOpdracht
           | 'EXIT' ]
```

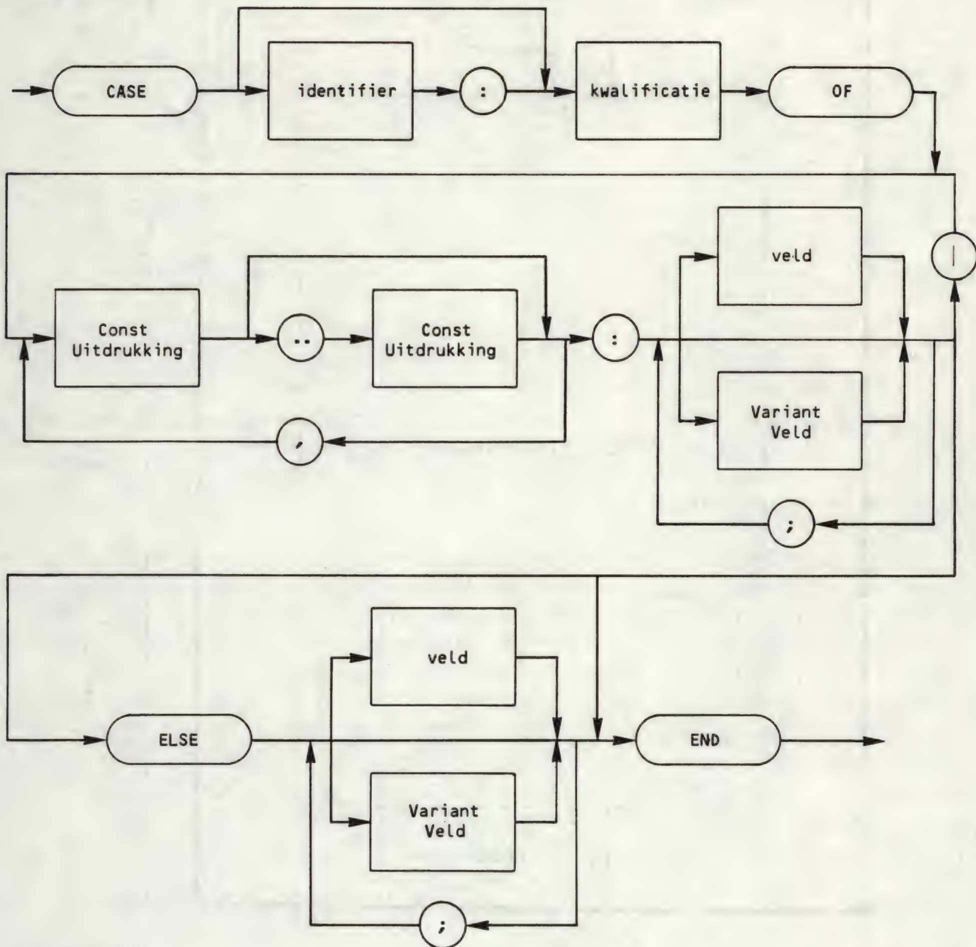
In het volgende hoofdstuk worden hieraan nog twee opdrachten toegevoegd.

Records met varianten

De componenten van een recordtype hoeven niet tot hetzelfde type te behoren. Bij records beschikken we over nog meer vrijheid : van een recordtype kunnen verschillende varianten voorkomen. Algemeen bestaat een record uit nul of meer vaste delen en nul of meer variabele delen. Een vast deel bevat de componenten die voor alle records gemeenschappelijk zijn. Een variabel deel bevat de beschrijving van de alternatieven voor de verschillende varianten. Bij elk variant deel behoort een etiketveld. De waarde van het etiketveld bepaalt de keuze van een variant deel.

De syntaxis voor records met varianten is :

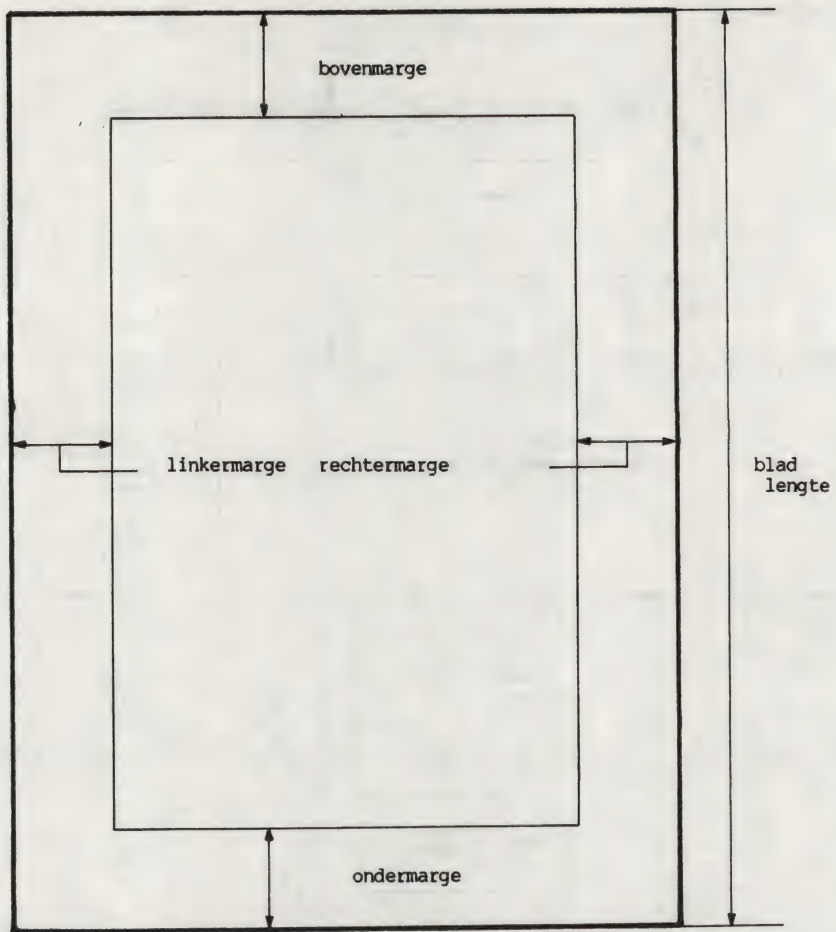
```
VariantVeldLijst  = 'CASE' [ identifier ] ':' kwalificatie 'OF'
                       variant
                    { | variant }
                    [ 'ELSE' VeldenLijst ]
                    'END'
variant           = [ CaseWaardenLijst ':' VeldenLijst ]
CaseWaardenLijst  = CaseWaarden { ',' CaseWaarden }
CaseWaarden       = ConstUitdrukking
                    [ '..' ConstUitdrukking ]
```



Voorbeeld :

In een systeem voor documentverwerking wordt elk document bladzijde voor bladzijde afgedrukt. Voor elke bladzijde is een opmaakstructuur gedefinieerd. Enkele belangrijke elementen voor de opmaak van een blad zijn : de afmetingen van een bladzijde, de boven-, onder-, linker- en rechtermarge.

De lengte van een bladzijde en de boven- en ondermarge worden meestal uitgedrukt in een aantal regels. Een regel wil zeggen een bepaalde verticale verplaatsing van de afdrukeenheid op de bladzijde. De kleinste eenheid voor deze verplaatsing is afhankelijk van de nauwkeurigheid van de afdrukeenheid. Sommige afdrukeenheden, bijvoorbeeld laserprinters, kunnen worden ingesteld tot op 1/400 inch. In het voorbeeld komt een regel overeen met de typische maat 1/6 inch. Deze laatste instelling stemt overeen met de gewone lettergrootte voor het type CHAR.

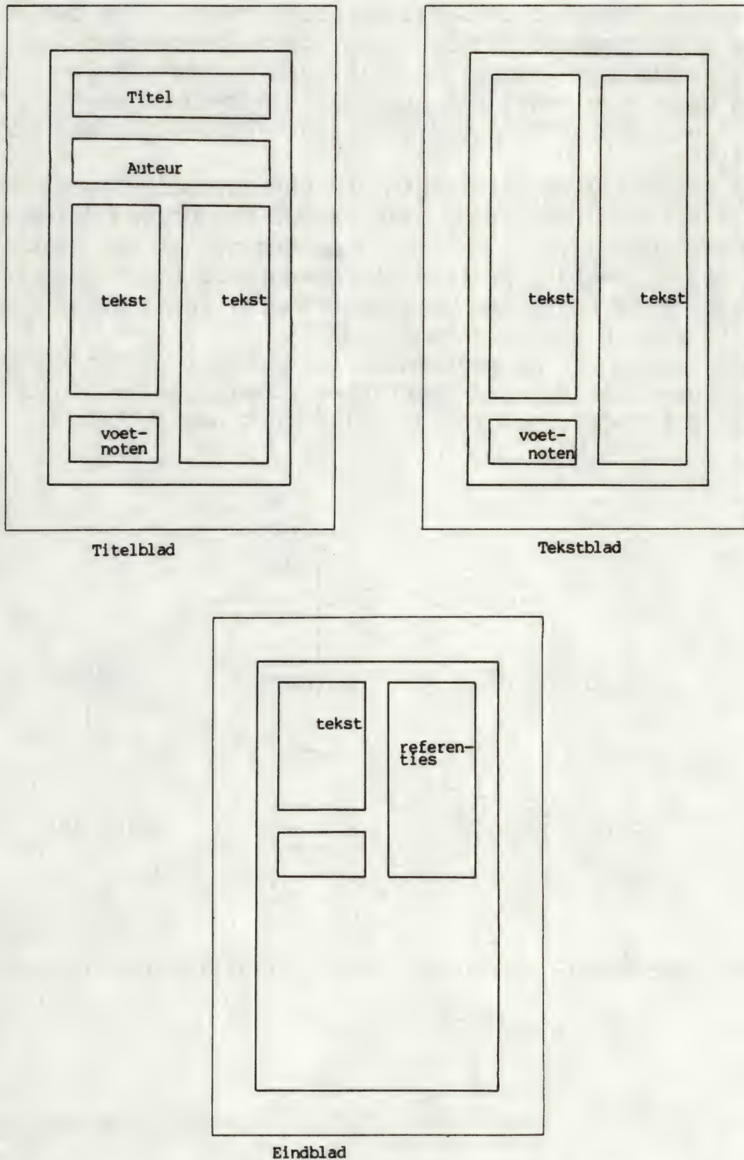


figuur 5.1 De opmaakelementen van een bladzijde

We definiëren nu een eenvoudige gegevensstructuur zodat we een bladzijde als een logisch object in een programma kunnen behandelen :

```
CONST maxRegel  = 60;
      maxKolom  = 65;
TYPE  Regel     = [0..maxRegel];
      Kolom     = [0..maxKolom];
      Bladzijde = RECORD
        bladlengte, bovenmarge, ondermarge : Regel;
        linkermarge, rechtermarge          : Kolom;
        tekst : ARRAY Regel,Kolom OF CHAR
      END;
```

Deze gegevensstructuur is geschikt voor een eenvoudige opmaakstructuur van teksten, bijvoorbeeld de bladzijden in een roman. Voor de opmaak van een artikel is deze gegevensstructuur te eenvoudig. We beschouwen nu een mogelijke opmaakstructuur voor een artikel. Een voorbeeld van zo'n artikel is weergegeven in de volgende figuur.

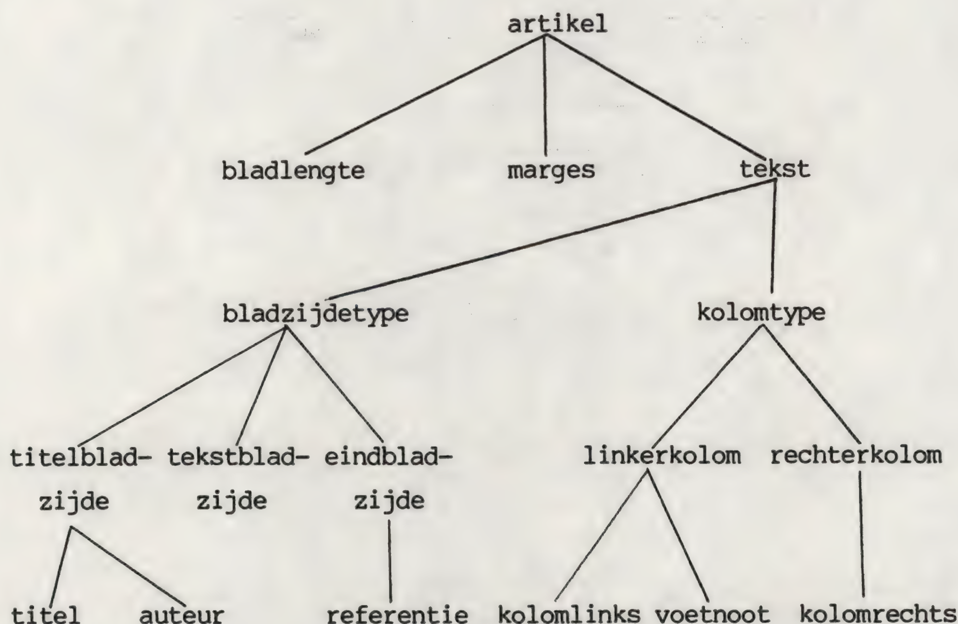


figuur 5.2 De opmaak van een artikel

Een artikel bestaat uit een of meer bladzijden met elk een verschillende opmaakstructuur. Voor het gemak veronderstellen we dat we in elk artikel de volgende typen bladzijden terugvinden : een titelblad, een tekstblad en een afsluitblad. Het titelblad heeft de volgende opmaak : de titel, de namen van de auteurs, een samenvatting en een tekstdeel. Het tekstdeel is opgesplitst in twee kolommen. Elke kolom bevat een deel tekst. Eventuele voetnoten zijn opgenomen in de linkerkolom; de ruimte voor de voetnoten is dus ten hoogste de helft van een tekstkolom. Een tekstblad bevat een tekstdeel opgemaakt in twee kolommen. Tot slot bevat het afsluitblad ook weer een tekstdeel en een referentielijst.

Deze beschrijving is slechts een vereenvoudigd model voor de opmaak van artikelbladzijden. Elk van de beschreven delen kan nog verder worden opgedeeld in kleinere eenheden, elk met eigen kenmerken. Bijvoorbeeld : de tekst is samengesteld uit paragrafen, met een paragraaftitel, een volgordekenmerk; in de tekst kunnen ook nog figuren of schema's voorkomen.

We definiëren nu de gegevensstructuur voor de opmaak van een artikel volgens het zojuist beschreven model. Ter verduidelijking stellen we dit model voor met de volgende boomstructuur :



figuur 5.3 De opmaakstructuur voor een artikel

```
CONST maxRegel          = 60;
      maxKolom          = 65;
TYPE  Regel             = [0..maxRegel];
      Kolom             = [0..maxKolom];
      BreedteTekst      = [0..maxKolom DIV 2];
      BladzijdeType     = (titelblad, tekstblad, eindblad);
      TekstKolom        = ARRAY Regel, BreedteTekst OF CHAR;
      KolomType         = (linkerkolom, rechterkolom);
      VoetnootGebied    = ARRAY [0..maxRegel DIV 2],BreedteTekst
                          OF CHAR;
      ReferentieGebied = TekstKolom;

      Bladzijde         = RECORD
        bladlengte, bovenmarge, ondermarge : Regel;
        linkermarge, rechtermarge          : Kolom;
        tekst : RECORD
          CASE bladzijde : BladzijdeType OF
            titelblad :
              titel  : ARRAY [1..2], Kolom OF CHAR;
              auteur : ARRAY Kolom OF CHAR
          | tekstblad :
          | eindblad :
              referentie : ReferentieGebied
          END;
          CASE kolom : KolomType OF
            linkerkolom :
              kolomLinks : TekstKolom;
              voetnoot : VoetnootGebied
          | rechterkolom :
              kolomRechts : TekstKolom
          END
        END
      END
```

5.3 Het verzamelingtype SET

Elk type definieert een eindige verzameling waarden; zo definieert bijvoorbeeld het type CARDINAL de waarden nul tot MAX(CARDINAL). In Modula-2 is een SET een verzameling van een aantal objecten met hetzelfde basistype. Een verzamelingtype wordt gedefinieerd volgens de syntaxis :

```
VerzamelingType = 'SET' 'OF' EnkelvoudigType
EnkelvoudigType = Enumeratietype | DeelintervalType | kwalificatie
```

VerzamelingType

De waarde van een verzameling wordt bepaald door de specificatie van de elementen :

```
verzameling = [ kwalificatie ] '{' [ element {',' element } ] '}'
element     = ConstUitdrukking [ '..' ConstUitdrukking ]
```

VerzamelingConstante

Een verzameling is ook een alternatief voor een factor in een uitdrukking :

```
factor = getal | verzameling | '(' uitdrukking ')' | 'NOT' factor
```

Het type

TYPE S = SET OF [Ø..1]

definieert het verzamelingtype S. De mogelijke waarden voor dit type zijn de verzamelingen

{} {Ø} {1} {Ø,1}

{} is de lege verzameling. Een element kan slechts één keer tot een verzameling behoren. Als het basistype n verschillende waarden heeft, dan heeft het verzamelingtype 2^n waarden.

Voor de verzamelingen gelden de volgende beperkingen :
- de verzameling bevat alleen constanten;
- het basistype is steeds een enumeratietype of een deelintervaltype. Het aantal mogelijke waarden van de elementen is meestal beperkt tot de woordlengte van de computer.

Voorbeelden :
De verzameling

{'A'..'D'}

bevat de elementen 'A', 'B', 'C' en 'D'.

De verzameling

{'A','E'..'H','K'}

bevat de elementen 'A', 'E', 'F', 'G', 'H' en 'K'.

De diskettestations van een personal computer worden aangeroepen met een symbolisch adres. Voor dit adres worden de letters 'A', 'B', 'C' tot en met 'P' gebruikt. Op het computersysteem is een verzameling stations aangesloten. We definiëren het type Aandrijving :

```
TYPE AdresAandrijving = ['A'..'P'];
     Aandrijving = SET OF AdresAandrijving;
```

Tijdens de werking van het systeem wordt de toestand van de stations bijgehouden. Een station is op een bepaald ogenblik actief indien een diskette in de aandrijving is geplaatst en het computersysteem de aanwezigheid ervan heeft vastgesteld. We houden de toestand van de stations bij met de variabele

```
VAR actiefStation : Aandrijving;
```

Als een station wordt geactiveerd, wordt het stationadres toegevoegd aan de verzameling actiefStation. Als de variabele actiefStation de waarde

{'A', 'B', 'D'}

heeft, betekent dit dat de stations 'A', 'B' en 'D' een diskette bevatten en door een programma kunnen worden gebruikt. Een lege verzameling geeft aan dat het systeem geen toegang heeft tot een station.
De opdrachten voor een menugestuurd systeem worden ingevoerd via een functietoets :

```
TYPE Functietoets = (F1, F2, F3, F4, F5, F6, F7, F8, F9, F1Ø)
```

Voor een toepassing definiëren we een opdrachtenverzameling

```
TYPE OpdrachtVerzameling = SET OF Functietoets;
VAR  opdracht : OpdrachtVerzameling
```

De waarde

{F1, F3..F5}

betekent dat op een bepaald ogenblik de opdrachten F1, F3, F4 en F5 kunnen worden uitgevoerd.

De kwalificatie van een verzameling in een uitdrukking

De operanden van een uitdrukking moeten altijd tot hetzelfde type behoren. Deze beperking geldt ook voor de waarden van het verzamelingtype. Voor de waarde van een verzameling kunnen we echter het type niet altijd ondubbelzinnig afleiden. Gegeven zijn de volgende definities :

```
TYPE Letters       = SET OF CHAR;
     KleineLetters = SET OF ['a'..'z'];
```

Het type van de verzameling

{'a','x'..'z'}

voldoet aan beide verzamelingtypen. Om dubbelzinnigheid te vermijden kwalificeren we de waarde van de verzameling :

Letters{'a', 'x'..'z'}

De kwalificatie van de waarde van een verzameling is steeds verplicht, behalve voor BITSET-verzamelingen. Het type BITSET wordt in een volgende paragraaf beschreven.

Bewerkingen met verzamelingen

We beschouwen de verzamelingen a en b :

```
TYPE CharSet = SET OF CHAR;
VAR a, b, c  : CharSet;
```

Met een toekenningsopdracht geven we a en b een waarde :

```
a := CharSet{'a', 'b', 'z'};
b := CharSet{'a', 'y', 'z'};
```

Op de verzamelingen a en b zijn de volgende bewerkingen gedefinieerd :

a + b : de vereniging

De vereniging van a en b bestaat uit alle elementen die of in a, of in b, of in beide verzamelingen voorkomen.
a + b heeft de waarde CharSet{'a', 'b', 'y', 'z'}.

a * b : de doorsnede

De doorsnede van a en b bestaat uit alle elementen die zowel in a als in b voorkomen.
a * b heeft de waarde CharSet{'a', 'z'}.

a - b : het verschil

Het verschil van a en b bestaat uit de elementen die wel in a maar niet in b voorkomen.
a - b heeft de waarde CharSet{'b'} en b - a heeft de waarde CharSet{'y'}.

a / b : het symmetrisch verschil

Het symmetrisch verschil bestaat uit de vereniging van de elementen die wel in a maar niet in b voorkomen en van de elementen die wel in b maar niet in a voorkomen. De uitdrukking

a / b

is identiek aan de uitdrukking

(a - b) + (b - a)

a / b heeft de waarde CharSet{'b', 'y'}.

Met de bewerkingen 'vereniging' en 'verschil' kunnen we uitsluitend een constante verzamelingwaarde toevoegen aan of weglaten uit een verzameling :

a + CharSet{'s'} geeft de waarde CharSet{'a', 'b', 'z', 's'}

a - CharSet{'b'} geeft de waarde CharSet{'a', 'z'}

a + CharSet{'x'..'z'} geeft de waarde CharSet{'a','b','x','y','z'}

Daarnaast verschaft Modula-2 nog de twee functies INCL en EXCL om de waarde van een uitdrukking toe te voegen aan of te verwijderen uit een verzameling. Gegeven is de variabele s met een verzamelingtype en x een uitdrukking waarvan het resultaattype overeenkomt met het basistype van s.

INCL(s,x) voegt de waarde van x toe aan de verzameling s;
EXCL(s,x) verwijdert de waarde van x uit de verzameling s

Voorbeeld :
Beschouw c = CharSet{'Ø'..'9'} en teken = 'a'.
Door de opdracht

INCL(c, CAP(teken))

krijgt c de waarde CharSet{'Ø'..'9', 'A'}.

Relaties

Op verzamelingen zijn ook een aantal relationele bewerkingen gedefinieerd :

De vergelijking

{1, 2, 3, 4} = {1..4}

heeft de waarde TRUE, en

a = b

met a = CharSet{'a', 'b', 'z'} en b = CharSet{'a', 'y', 'z'} heeft de waarde FALSE. De vergelijking

a # b

levert de waarde TRUE.
De operatoren '<=' en '>=' worden gebruikt voor bewerkingen op deelverzamelingen.

a <= b

levert de waarde TRUE als elk element van a ook een element is van b; anders is de waarde FALSE. Als a <= b geldt, is a een deelverzameling van b.

a >= b

levert de waarde TRUE als elk element van b ook een element is van a; anders is de waarde FALSE. Als a >= b geldt, is b een deelverzameling van a.

Met de operator IN gaan we na of een element tot een verzameling behoort :

'a' IN a

levert de waarde TRUE, en

'd' IN a

de waarde FALSE voor a = CharSet{'a', 'b', 'z'}.

Opmerking : De bewerkingen '<' en '>', om na te gaan of een deelverzameling een echte deelverzameling is, zijn niet gedefinieerd. De verzameling a is een echte deelverzameling van b als elk element van a ook een element is van b en bovendien a # b geldt. We testen dit daarom met :

(a <= b) AND (a # b).

5.4 Het standaardtype BITSET

De definitie van het type BITSET is

BITSET = SET OF [0..W-1]

Hierbij is W afhankelijk van de woordlengte van het computersysteem. Bij kleinere computers is de waarde van W dikwijls 16, bij grotere systemen 32 of zelfs meer. Het type BITSET is een standaardtype zoals de typen CARDINAL, INTEGER, CHAR, BOOLEAN en REAL. Voor het type BITSET zijn dezelfde bewerkingen gedefinieerd als voor de verzamelingen. De kwalificatie van de waarde van een BITSET-verzameling is facultatief.

Met het type BITSET kunnen bewerkingen op het laagste niveau, het bitniveau, van de computer worden uitgevoerd. Voor een gegeven computersysteem wordt een BITSET voorgesteld met een woord van twee bytes. Als we de elementen van de verzameling nummeren van 0 tot 15, is de geheugenvoorstelling :

```
7  6  5  4  3  2  1  0      15 14 13 12 11 10 9 8
   laagste byte                 hoogste byte
```

Voor elk van de volgende waarden wordt het bitpatroon afgedrukt :

BITSET{}	00000000 00000000
BITSET{7,5,3,1,15,13}	10101010 10100000
BITSET{0..15}	11111111 11111111
BITSET{7}	10000000 00000000

Een belangrijke toepassing van het type BITSET is het gebruik van een 'bitmap'. Een bitmap is een zeer compacte geheugenvoorstelling voor het aangeven van de aan- of afwezigheid van informatie. We geven hiervan twee voorbeelden :

- het externe geheugen van een computer bevat een of meer magneetschijven. Met een 'file-systeem' worden de bestanden op deze magneetschijven afgebeeld. Elke magneetschijf is verdeeld in een aantal concentrische sporen, elk spoor is weer verdeeld in een aantal sectoren. Het laagste niveau van een filesysteem zorgt voor het beheer van deze sectoren. Met elke sector kan een bit uit een bitmap overeenstemmen zodat het filesysteem snel kan nagaan of een sector reeds in gebruik is;
- bij een grafisch uitvoerapparaat, bijvoorbeeld een grafisch beeldscherm of een laserprinter, wordt de informatie afgebeeld in een raster van puntjes. Met elk punt van het raster komt een bit in het geheugen overeen. Een bitwaarde '1' betekent bijvoorbeeld zwart, de waarde 'Ø' wit. De verzameling van deze bits vormt weer een bitmap. Het aantal elementen van de bitmap is afhankelijk van de resolutie van het uitvoerapparaat. De resolutie is de maat voor het aantal punten per lengte-eenheid, bijvoorbeeld 3ØØ dpi (dots per inch). We definiëren een bitmap voor de afbeelding van een teken :

```
TYPE TekenBitmap = ARRAY [1..2],[1..32] OF BITSET;
```

De bitvoorstelling voor een pijltje is bijvoorbeeld :

```
00000000000000001000000000000000
00000000000000001100000000000000
00000000000000001110000000000000
00000000000000001111000000000000
00000000000000001111100000000000
00000000000000001111110000000000
00000000000000001111111000000000
00000000000000001111111100000000
00000000000000001111111110000000
11111111111111111111111111000000
11111111111111111111111111100000
11111111111111111111111111110000
11111111111111111111111111111000
11111111111111111111111111111100
11111111111111111111111111111110
11111111111111111111111111111111
11111111111111111111111111111111
11111111111111111111111111111110
11111111111111111111111111111100
11111111111111111111111111111000
11111111111111111111111111110000
11111111111111111111111111100000
11111111111111111111111111000000
00000000000000001111111110000000
00000000000000001111111100000000
00000000000000001111111000000000
00000000000000001111110000000000
00000000000000001111100000000000
00000000000000001111000000000000
00000000000000001110000000000000
00000000000000001100000000000000
00000000000000001000000000000000
```

figuur 5.4 De bitmap-voorstelling voor een pijl

Het sorteren van een reeks getallen

Stel we hebben een programma voor het van klein naar groot sorteren van natuurlijke getallen, die tussen 1 en 27000 liggen. Deze getallen worden van een bestand gelezen. Indien een getal meerdere keren voorkomt wordt dit als een fatale fout beschouwd. De getallen hebben geen speciale betekenis. De uitvoer is een lijst met de ingevoerde getallen in stijgende volgorde. Voor de uitvoering van dit programma gelden de volgende beperkingen :

in het interne geheugen is slechts ruimte voor ongeveer 1700 woorden van 16 bits en de uitvoering van het programma mag slechts enkele ogenblikken duren.

We geven voor dit probleem een eerste oplossing waarbij we gebruik maken van de eigenschappen van de invoerstroom en van de werkomgeving. Het invoerbestand wordt 27 keer gelezen. Bij de eerste stap selecteren we alle getallen tussen 1 en 1000, bij de volgende stap alle getallen tussen 1001 en 2000. We slaan de waarden op in een tabel in het geheugen. Hierbij kunnen we twee strategieën volgen :
(1) we plaatsen de getallen in volgorde van binnenkomst in de tabel en we sorteren de tabel met een sorteerprogramma. We onthouden hierbij het aantal geselecteerde getallen;
(2) we wissen alle elementen van de tabel door het toekennen van de waarde nul. Daarna berekenen we voor elk geselecteerd getal een plaats in de tabel zodat de getallen automatisch in de juiste volgorde worden bewaard.

We lossen het probleem op volgens de tweede strategie. In het programma gebruiken we de standaardin- en -uitvoerbestanden. Als de invoer beschikbaar is op een bestand op een extern geheugen, dan kan dit met behulp van 'redirection' aan de standaardinvoer worden gekoppeld. Noemen we de opdracht 'sorteer' en het bestand 'getallen', dan wordt dit bestand in sommige besturingssystemen met de standaardinvoer gekoppeld met de opdracht

```
sorteer < getallen
```

Voor dit probleem is deze oplossing niet geschikt : we moeten immers het invoerbestand steeds meer dan één keer doorlopen willen we het bestand opnieuw vanaf het begin lezen. We gebruiken het volgende algoritme :

(1) lees de naam van het bestand;
(2) open het invoerbestand;
(3) koppel het bestand aan de standaardinvoer.

Bij elke doorgang herhalen we de volgende opdrachten :

(4) positioneer de eerste leesopdracht bij het begin van het bestand;
(5) lees het bestand.

Tot slot wordt het bestand gesloten.

We geven nu een opsomming van de gegevensstructuren en de bewerkingen die we voor het oplossen van dit probleem nodig hebben. Deze objecten worden in een later hoofdstuk nader bekeken. Het abstracte datatype Files definieert de gegevensstructuur File en de bewerkingen hierop.

Bij het openen kunnen we aan een bestand (file) de volgende kenmerken toekennen :

- BinTextMode = (binMode, textMode) :
  Het bestand wordt beschouwd als een binair bestand of als een tekstbestand. In een binair bestand wordt alle informatie in machinecode bewaard. In een tekstbestand wordt de informatie als tekst bewaard. Voor het gegeven probleem is de invoer een tekstbestand.

- ReadWriteMode = (readOnly, readWrite, appendOnly) :
  Uit het bestand kan alleen worden gelezen (readOnly), er kan ook worden weggeschreven naar het bestand (readWrite) of het bestand kan alleen worden aangevuld (appendOnly). Voor bovenstaand probleem wordt uit het bestand uitsluitend gelezen.

- FileState = (ok, nameError, noFile, existingFile, deviceError, noMoreRoom, accessError, notOpen, endError, outsideFile, otherError) :
  Dit kenmerk verschaft informatie over de toestand van het bestand. De toestand is in orde ('ok'), indien het bestand bij het openen bestaat. We veronderstellen dat het invoerbestand inderdaad bestaat, zodat we met de toestand geen rekening hoeven te houden.

We gebruiken de volgende variabelen :

```
VAR naam      : ARRAY [Ø..13] OF CHAR;
    f         : File;
    toestand  : FileState;
```

We openen het tekstbestand met de opdrachten :

```
ReadString(naam); (* lees de naam van het bestand *)
Open(f, naam, textMode, readOnly, toestand);
```

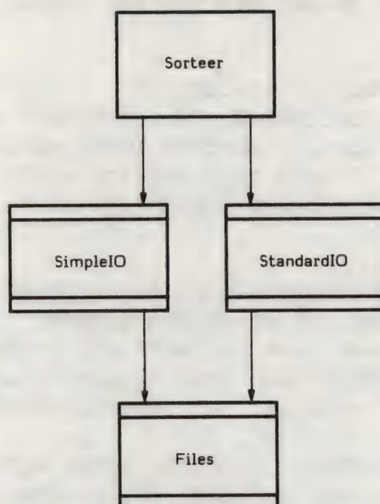
Een bestand wordt bij het begin gepositioneerd met de opdracht

```
Reset(f, toestand);
```

Het abstracte datatype StandardIO levert de bewerkingen voor de koppeling van een bestand met de standaardin- en -uitvoer. We gebruiken de bewerking SetInput :

```
SetInput(f)
```

koppelt het bestand f aan de standaardinvoer. Voor het oplossen van het probleem geldt het volgende abstractieschema :



figuur 5.5 Sorteren met 'redirection'

We geven nu de programmamodule voor het sorteren van het invoerbestand met natuurlijke getallen :

```
MODULE Sorteer;
FROM SimpleIO   IMPORT WriteCard, WriteString, WriteLn, EOT,
                       ReadCard, ReadLn, ReadString;
FROM StandardIO IMPORT SetInput;
FROM Files      IMPORT File, FileState, Open, Reset, Close,
                       BinTextMode, ReadWriteMode;

CONST maxTabel = 1000;
      maxGetal = 27000;
VAR   tabel    : ARRAY [1..maxTabel] OF CARDINAL;
      i, j     : CARDINAL;
      getal    : CARDINAL;
      succes   : BOOLEAN;
      naam     : ARRAY [0..13] OF CHAR;
      f        : File;
      toestand : FileState;
```

```
BEGIN
(* instellen van de standaardinvoer *)
WriteString('Naam invoerbestand :');
ReadString(naam);ReadLn;
Open(f, naam, textMode, readOnly, toestand);
SetInput(f);

(* verwerking van een doorloop *)
j := 1;
ReadCard(getal, succes);
ReadLn;
WHILE (j <= maxGetal DIV maxTabel) AND NOT EOT() DO
  (* wis de tabel *)
  FOR i := 1 TO maxTabel DO
    tabel[i] := Ø
  END;

  (* verwerk het invoerbestand *)
  WHILE NOT EOT() DO
    IF (getal >= (j - 1) * maxTabel + 1) AND
       (getal <= j * maxTabel)
    THEN
      IF tabel[getal - (j - 1) * maxTabel] # Ø
      THEN
        WriteString('dubbel getal');
        RETURN
      END;
      tabel[getal - (j - 1) * maxTabel] := getal
    END;
    ReadCard(getal, succes);ReadLn
  END;

  (* druk de elementen van de tabel af *)
  FOR i := 1 TO maxTabel DO
    IF tabel[i] # Ø
    THEN
      WriteCard(tabel[i],5);WriteLn
    END
  END;

  (* initieer de volgende stap *)
  Reset(f,toestand);
  ReadCard(getal, succes);ReadLn
  INC(j)
END;

(* sluit invoerbestand af *)
Close(f,toestand)
END Sorteer.
```

Een alternatieve oplossing

Deze oplossing heeft een aantal nadelen : het invoerbestand moet herhaaldelijk worden gelezen en we kunnen daarom geen gebruik maken van de standaardtechnieken voor het koppelen van de standaardinvoer. We gebruiken echter niet alle specifieke kenmerken van de invoerstroom (natuurlijke getallen, alle verschillend en behorend tot het interval 1 tot 27000). Voor het oplossen is het voldoende te specificeren welke getallen in dit interval voorkomen. We stellen het invoerbestand in het geheugen voor als een bitmap van 27000 bits. We lezen de invoer een keer. Bit_i wordt '1' indien het getal i in de invoer voorkomt. Tot slot doorlopen we de bitmap en we drukken de positie af van elk '1'-bit.
Dit geeft het volgende algoritme :

```
bitmap : ARRAY [1..27000] OF BIT;

(* initiëren bitmap *)
FOR i := 1 TO 27000 DO bitmap[i] := 0 END

(* voeg elk getal toe aan de bitmap *)
Stel voor elk getal i in het bestand bitmap[i] := 1

(* uitvoer van de gesorteerde reeks *)
FOR i := 1 TO 27000 DO
  IF bitmap[i] = 1 THEN druk i af END
END
```

De bitmap stellen we voor met een tabel van BITSET-elementen :

```
CONST maxGetal    = 27000;
      woordLengte = 16;
      maxTabel    = maxGetal DIV woordLengte + 1;
VAR   bitmap      : ARRAY [1..maxTabel] OF BITSET;
```

Voor de afbeelding van een getal g op een element van de bitmap berekenen we eerst de index van het BITSET-element en daarna het volgnummer van de bit :

```
p := getal DIV woordLengte + 1;        (* index BITSET-element  *)
q := getal MOD woordLengte;     (* volgnummer van de bit *)
```

De gewenste bit voegen we toe met de opdracht

```
INCL(bitmap[p],q)
```

```
MODULE Sorteer;
FROM SimpleIO IMPORT ReadCard, EOT,
                     WriteCard, WriteString, WriteLn;

CONST maxGetal      = 27000;
      woordLengte   = 16;
      maxTabel      = maxGetal DIV woordLengte + 1;
VAR   bitmap        : ARRAY [1..maxTabel] OF BITSET;
      i, j, p, q    : CARDINAL;
      getal         : CARDINAL;
      succes        : BOOLEAN;

BEGIN
(* initieer de bitmap *)
FOR i := 1 TO maxTabel DO
  bitmap[i] := BITSET{}
END;

(* lees het invoerbestand *)
LOOP
  ReadCard(getal, succes);
  ReadLn;
  IF EOT() THEN EXIT END;
  p := getal DIV woordLengte + 1;
  q := getal MOD woordLengte;
  IF q IN bitmap[p]
  THEN
    WriteString('dubbel getal');
    RETURN
  END;
  INCL(bitmap[p],q)
END;

(* druk de gesorteerde reeks af *)
FOR i := 1 TO maxTabel DO
  FOR j := 0 TO woordLengte - 1 DO
    IF j IN bitmap[i]
    THEN
      WriteCard((i - 1) * woordLengte + j, 5);
      WriteLn
    END
  END
END

END Sorteer.
```

Literatuur :

Bentley J., 'Programming Pearls, Cracking the Oyster', Communications of ACM, augustus en oktober 1983

Bentley J., 'Programming Pearls, Writing Correct Programs', Communications of ACM, december 1983

Bentley J., 'Programming Pearls, How to Sort', Communications of ACM, april 1984

Caillieu R., 'How to Avoid Getting SCHLONKED by Pascal', Sigplan, december 1982

Modula-2 Standard Library Definition Modules, Modula-2 News, januari 1985

Spector D., 'Ambiguities and Insecurities in Modula-2', Sigplan, augustus 1982

Wirth N., 'Programming in Modula-2', derde verbeterde druk, Springer-Verlag, Berlijn 1985

Oefeningen

1. Uit hoeveel elementen is een tabel samengesteld voor elk van de volgende indextypen :

a. [1..5]
b. [1..10]
c. [-5..10]
d. [-5..5] OF ARRAY [-5..5]
e. [1..50] OF ARRAY [1..10] OF ARRAY [1..25]

2. Bereken de uitdrukking :
3 IN ({1, 2, 3} * ({2, 3, 4} - {3, 4, 5}))

3. Hoeveel elementen kan een verzameling maximaal bevatten voor je Modula-2 implementatie ?

4. De invoer voor een programma is :
- een getal n met type CARDINAL;
- n gehele getallen.
De uitvoer bestaat uit de n getallen in omgekeerde volgorde. Schrijf hiervoor een programmamodule. Test het programma voor n = 0, n = 1 en n = 10.

5. De invoer voor een programma is :
- een getal n met type CARDINAL;
- n gehele getallen.
De uitvoer bestaat uit de n getallen die 'geroteerd' worden afgedrukt. Dit betekent : het tweede gelezen getal wordt, indien aanwezig, eerst afgedrukt. Het derde gelezen getal wordt daarna afgedrukt enzovoort voor de overige getallen. Het eerste gelezen getal wordt als laatste afgedrukt. Test het programma met n = Ø, n = 1 en n = 1Ø.

6. Wijzig de oplossing voor opgave 5 zodat de ingelezen reeks getallen over een variabel aantal elementen r wordt geroteerd.
a. Formuleer een oplossing met een hulptabel;
b. Formuleer een oplossing waarbij je slechts over enkele geheugenwoorden kan beschikken.

7. De resultaten van een examen worden ingedeeld volgens de volgende graden :

behaalde punten	graad
1ØØ - 9Ø	A
89 - 8Ø	B
79 - 7Ø	C
69 - 6Ø	D
59 - Ø	E

De invoer van een programma is een bestand met per regel het behaalde aantal punten per student.
a. tel het aantal studenten per graad;
b. bereken het rekenkundig gemiddelde per graad;
c. tel het totaal aantal studenten;
d. bereken het rekenkundig gemiddelde en de standaardafwijking voor alle studenten.

8. De antwoorden van een enquête met ja-neen-vragen worden als volgt geregistreerd : voor elke ondervraagde persoon bevat het invoerbestand een regel met een identificatienummer van vier cijfers en daarna de antwoorden op de vragen (1 = 'ja', 'Ø' = 'neen'). Het invoerbestand is :

```
ØØ23 Ø 1 1 Ø 1 Ø 1 1 Ø 1
Ø12Ø Ø 1 Ø 1 Ø 1 1 1 Ø Ø
Ø122 Ø 1 1 Ø 1 1 1 1 1 1
Ø235 1 1 Ø Ø 1 Ø Ø 1 1 1
Ø24Ø 1 1 1 1 1 1 1 1 1 1
```

```
0256 0 1 0 0 1 0 0 1 0 1
0500 1 0 1 0 1 0 1 0 1 0
0514 1 1 1 0 0 1 1 0 1 0
0580 0 0 0 0 0 0 0 0 0 0
0587 1 0 1 1 0 1 1 0 1 0
0590 0 1 0 0 1 1 0 0 0 1
0630 0 1 0 1 0 1 0 1 0 1
0634 1 0 1 0 1 1 0 1 0 1
```

De juiste antwoorden zijn :

```
                         0 1 0 0 1 0 0 1 0 1
```

Bereken voor elke ondervraagde het aantal correcte antwoorden en de graad die daarmee overeenkomt. De graden worden als volgt bepaald :

```
    hoogste score         : graad 'A'
    tweede hoogste score  : graad 'B'
    derde hoogste score   : graad 'C'
    vierde hoogste score  : graad 'D'
    overige scores        : graad 'E'
```

Druk voor elke deelnemer per regel af :

```
    identificatienummer  score  graad
```

9. Een programma leest n, met 1 <= n <= 20, elementen van twee tabellen x en y. Het type van een tabel is :

```
TYPE Index = [1..20];
     Tabel = ARRAY Index OF CARDINAL;
```

a. De overeenkomstige elementen van de tabellen worden met elkaar vergeleken. Definieer een tabeltype voor het opslaan van de resultaten van deze vergelijkingen;
b. lees de elementen van de tabellen x en y. Bepaal de elementen van tabel z met het type gedefinieerd in (a);
c. de uitvoer bestaat uit n regels met per regel een element van tabel x en van tabel y en een passende tekst 'kleiner dan', 'groter dan' of 'gelijk aan'.

10. De invoer van een programma is een getal n, met 0 <= n <= 100, en n cijfers, met '0' <= cijfer <= '9'. Tel het aantal paren van elk cijfer in de invoerstroom.

11. Bereken alle priemgetallen kleiner dan 10000. Gebruik de methode 'de zeef van Eratosthenes' :
- definieer een gegevensstructuur met 10000 elementen;
- het elementtype heeft de waarden 'priemgetal' of 'geen

priemgetal';
- initieer de tabel met alle elementwaarden = 'priemgetal';
- elimineer alle veelvouden van twee, drie enzovoort;
- druk de priemgetallen af.

Gebruik zo weinig mogelijk geheugenruimte voor de opslag van de gegevensstructuur.

12. De oplossing van de programmamodule Sorteer gebruikt voor een woordlengte van zestien bits 27000 DIV 16 = 1688 woorden. Wijzig het algoritme zodat de nodige geheugenruimte wordt herleid tot minder dan 1000 woorden.

13. Veronderstel dat elk getal in de invoerstroom van programma Sorteer hoogstens tien keer voorkomt. Wijzig het programma Sorteer waarbij de geheugenruimte beperkt is tot ongeveer 6K woorden.

6 Procedures

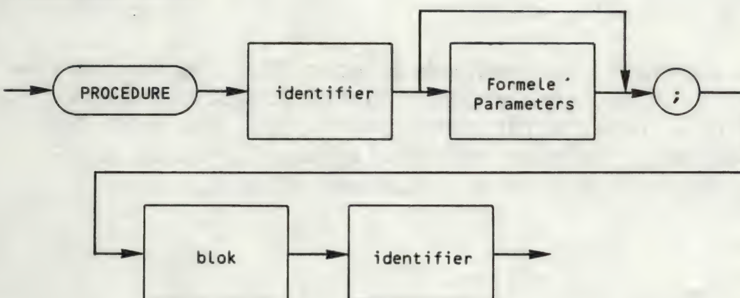
Bij het oplossen van een ingewikkeld probleem splitsen we dit in een aantal kleinere deelproblemen. De deeltaken die hieruit ontstaan zijn eenvoudiger en gemakkelijker te behandelen. Zodra we deze deeltaken hebben geïdentificeerd formuleren we de oplossing van het oorspronkelijke probleem in termen van de deeltaken. De algoritmen en de programma's voor deze deeltaken kunnen afzonderlijk worden ontwikkeld.

Modula-2 bevat drie typen subprogramma's voor de specificatie van deze deeltaken : procedures, functieprocedures en modulen. De modulen worden in een volgend hoofdstuk beschreven. Met een procedure of een functieprocedure kunnen we een willekeurig aantal (complexe) bewerkingen definiëren en daarna aanroepen met een identifier in een procedure-opdracht of in een uitdrukking. De procedure vormt aldus het basismechanisme voor functionele abstracties.

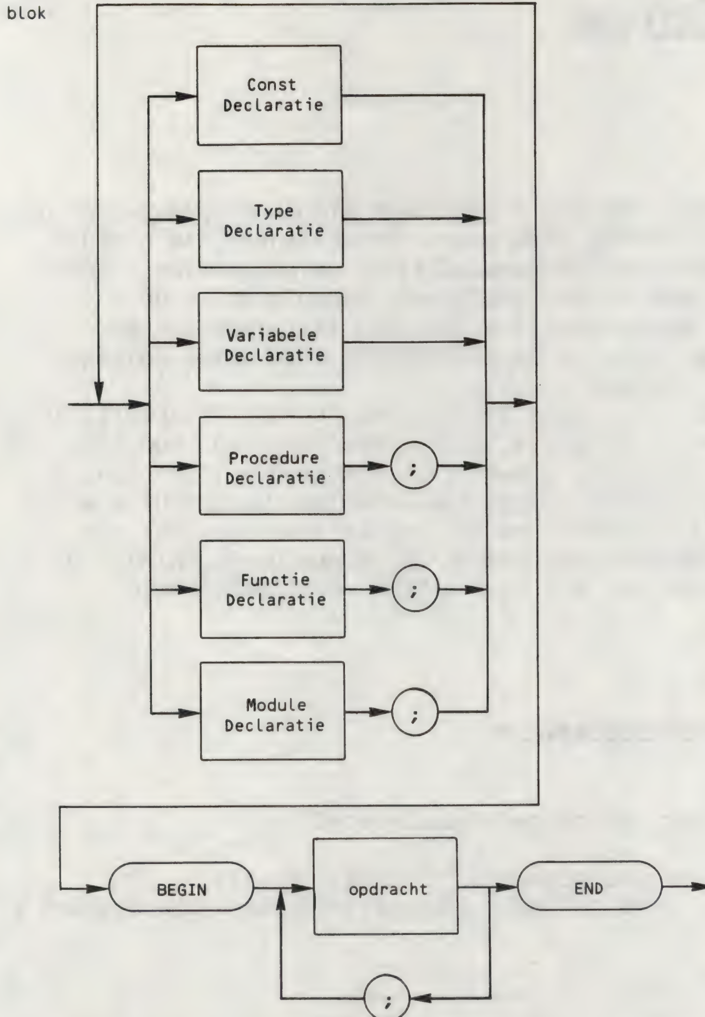
6.1 De proceduredeclaratie

De syntaxis voor een proceduredeclaratie is :

```
ProcedureDeclaratie = ProcedureKop ';' blok identifier
ProcedureKop        = 'PROCEDURE' identifier [ FormeleParameters ]
```



```
blok                   = { declaratie }
                         [ 'BEGIN' OpdrachtenRij ]
                         'END'
```



De procedurekop verbindt een naam aan de procedure die we declareren en zorgt ook voor de formele beschrijving van de parameters waarmee de procedure later wordt aangeroepen. De naam voor de identifier is willekeurig; voor de duidelijkheid laten we een procedurenaam steeds met een werkwoordsvorm beginnen, bijvoorbeeld :

ReadChar, ReadLn,
ZoekElement, DrukTabel.

De procedurekop bevat naar keuze ook een lijst met parameters, FormeleParameters. De betekenis en het gebruik van deze parameters wordt in de volgende paragraaf beschreven. De romp van een procedure, het blok, bestaat uit nul of meer declaraties en een facultatief opdrachtendeel. In het declaratiedeel worden alle objecten gedefinieerd die lokaal zijn voor het blok. Het blok is het enige deel in de programmatekst waar naar deze identifiers kan worden verwezen. Het opdrachtendeel begint met het symbool 'BEGIN' en eindigt met het symbool 'END'. Tussen deze beide symbolen wordt de opdrachtenrij gespecificeerd, zodat de gewenste bewerkingen worden uitgevoerd. De proceduredeclaratie eindigt met het herhalen van de identifier. Deze herhaling verschaft de compiler bijkomende controlemogelijkheden zodat kan worden of nagegaan dat de besturingsstructuren, en eventueel ook geneste procedures, steeds op de juiste wijze met een 'END'-symbool zijn afgesloten. We illustreren een proceduredeclaratie met de procedure ZoekMinimumTabel. De procedure zoekt het kleinste element van een tabel a :

```
CONST n   = 100;
VAR   a   : ARRAY [1..n] OF CARDINAL;
      min : CARDINAL.;

PROCEDURE ZoekMinimumTabel;
VAR i : CARDINAL;
BEGIN
min := a[1];
FOR i := 2 TO n DO
  IF a[i] < min
  THEN
    min := a[i]
  END
END
END ZoekMinimumTabel;
```

Parameters

In de procedurekop wordt eventueel ook een lijst met parameters, FormeleParameters, vermeld. De parameters hebben voor de procedure dezelfde betekenis als de operanden voor een rekenkundige operator : de parameters definiëren de identifiers van de objecten waarop bewerkingen door de procedure worden uitgevoerd. De parameters in de kopregel van de procedure zijn formele parameters : we gebruiken ze om de bewerkingen van de procedure te formuleren. Bij het uitvoeren van de procedure worden de waarden van de formele parameters gezet in een lijst met actuele parameters. Deze lijst met actuele parameters wordt vermeld bij het aanroepen van de procedure.

Parameterbinding

We onderscheiden twee mechanismen voor de koppeling van de actuele aan de formele parameters : de variabele-parameter en de waardeparameter. Bij de variabele-parameter is de actuele waarde steeds een variabele. In de procedure stemt een formele parameter dan ook overeen met een variabele. Over het algemeen levert de procedure-aanroep hiervoor het geheugenadres van de variabele. De procedure kan de waarde van de parameter raadplegen en wijzigen. Elke nieuwe waarde is automatisch ook bekend in het aanroepende programma. Het variabele-parametermechanisme gebruiken we steeds voor parameters waarvoor de procedure een waarde naar het aanroepende programma uitvoert (uitvoerparameter; output), of voor parameters waarvoor een waarde in de procedure wordt ingevoerd en eventueel ook een nieuwe waarde wordt uitgevoerd (in- en uitvoerparameter; input-output). We specificeren een variabele-parameter in de lijst met formele parameters als we de identifier laten voorafgaan door het symbool 'VAR'.

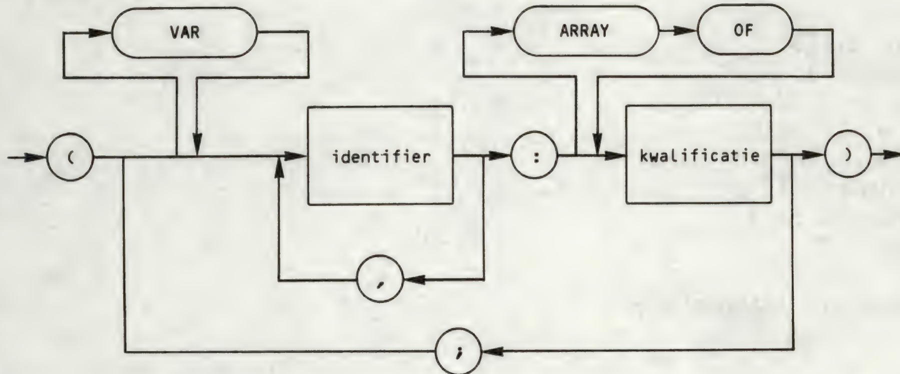
Bij de waardeparameter wordt alleen de waarde van de parameter aan de procedure meegedeeld. Meestal wordt deze waarde in een lokale geheugenruimte van de procedure gekopieerd. De actuele waarde van een waardeparameter is steeds een uitdrukking, bijvoorbeeld een constante, een variabele of een willekeurige uitdrukking. De procedure kan deze waarde raadplegen en eventueel ook wijzigen. In dit laatste geval wordt alleen de waarde van de lokale kopie gewijzigd. We gebruiken een waardeparameter als uitsluitend een waarde in de aangeroepen procedure wordt ingevoerd (invoerparameter; input). We specificeren een waardeparameter in de formele parameterlijst door de afwezigheid van het 'VAR'-symbool.

Opmerking : Bij de waardeparameter wordt voor de parameter een lokale geheugenruimte gereserveerd. Soms is deze parameter een datastructuur die uit talrijke elementen of velden bestaat. Om onnodig, en soms tijdrovend, kopiëren te vermijden, is voor omvangrijke datastructuren meestal een variabele-parameter aangewezen, zelfs als uitsluitend de waarde in de procedure wordt ingevoerd. We zijn dan zelf verantwoordelijk voor het juiste gebruik van deze datastructuur. Met commentaar kunnen we de betekenis van de parameters toelichten :

```
(* in *)    : invoerparameter;
(* uit *)   : uitvoerparameter;
(* inUit *) : in- en uitvoerparameter;
```

De syntaxis voor de lijst met formele parameters is :

```
FormeleParameters = '(' [ FPSectie { ';' FPSectie } ] ')'
FPSectie          = [ 'VAR '] IdentLijst ':' FormeelType
FormeelType       = kwalificatie
```



Voorbeelden :

We declareren opnieuw de procedure ZoekMinimumTabel maar nu met het gebruik van parameters. Hiervoor hebben we twee parameters nodig : een waardeparameter voor de tabel waarvoor het minimum wordt gezocht en een variabele-parameter voor de uitvoer van het resultaat.

```
CONST n     = 100;
TYPE  Tabel = ARRAY [1..n] OF CARDINAL;


PROCEDURE ZoekMinimumTabel(    a   : Tabel;
                           VAR min : CARDINAL);
VAR i : CARDINAL;

BEGIN
min := a[1];
FOR i := 2 TO n DO
  IF a[i] < min
  THEN
    min := a[i]
  END
END
END ZoekMinimumTabel;
```

In het tweede voorbeeld zoeken we het minimum en de plaats van dit minimum :

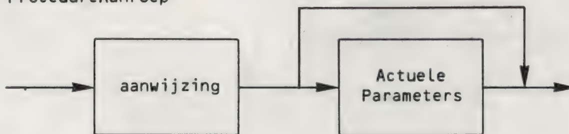
```
PROCEDURE ZoekMinPlaatsTabel(    a      : Tabel;
                             VAR min    : CARDINAL;
                             VAR plaats : CARDINAL);
VAR i : CARDINAL;

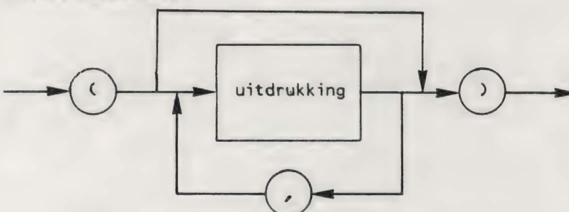
BEGIN
min := a[1];
plaats := 1;
FOR i := 2 TO n DO
  IF a[i] < min
  THEN
    min := a[i];
    plaats := i
  END
END
END ZoekMinPlaatsTabel;
```

6.2 De procedure-aanroep

Een procedure in een programma wordt aangeroepen met een procedure-aanroep. Deze procedure-aanroep bestaat uit de naam van de procedure eventueel gevolgd door de lijst met actuele parameters. De syntaxis voor de procedure-aanroep is :

```
ProcedureAanroep  = aanwijzing [ ActueleParameters ]
ActueleParameters = '(' [ UitdLijst ] ')'
```





De koppeling van de actuele parameters met de formele parameters gaat als volgt : de eerste actuele parameter wordt gekoppeld aan de eerste formele parameter, de tweede actuele aan de tweede formele, enzovoort. Voor de koppeling gelden de regels :
- het aantal actuele parameters is gelijk aan het aantal formele;
- voor een waardeparameter is de actuele parameter een uitdrukking. De waarde van de uitdrukking moet qua type passen bij dat van de formele parameter;
- voor een variabele parameter is de actuele parameter een variabele. Het type van deze variabele is hetzelfde als het type van de formele parameter.

Voorbeeld :

```
CONST n        = 100;
TYPE Tabel     = ARRAY [1..n] OF CARDINAL;
VAR  a, b      : Tabel;
     minimumA  : CARDINAL;
     minimumB  : CARDINAL;

...
ZoekMinimumTabel(a, minimumA);
ZoekMinimumTabel(b, minimumB)
```

De aanroep van ZoekMinimumTabel voor de tabellen a en b illustreert het nut van de parameters : we roepen een procedure aan op verschillende plaatsen in een programma en tevens voor verschillende waarden en variabelen. Met behulp van procedures en gebruik makend van parameters schrijven we aldus stukken code die overal in een programma kunnen worden aangeroepen.
Het uitvoeren van een procedure heeft het volgende effect :
- de formele parameters worden beschouwd als variabelen die lokaal in de procedure zijn gedefinieerd. De waardeparameters krijgen als beginwaarde de waarde van de actuele parameter. De variabele-parameters worden tijdens de uitvoering van de procedure vervangen door de overeenkomstige actuele parameters;
- de opdrachten van de procedurerompworden uitgevoerd.

6.3 Open-array-parameters

We beschouwen de volgende declaraties :

```
TYPE Tabel1 = ARRAY [1..n] OF CARDINAL;
     Tabel2 = ARRAY [1..m] OF CARDINAL;
```

Gevraagd wordt een procedure te maken voor het zoeken van het minimum in een tabel, die geschikt is voor beide type tabellen.

We hebben een algemene procedure nodig, waarvoor een procedurekop in de vorm

```
PROCEDURE ZoekMinTabel(    a       : AlgemeneTabel;
                       VAR minimum : CARDINAL);
```

zowel voor het type Tabell als voor Tabel2 moet voldoen. Het type van de formele parameter moet altijd wat toekenningsopdrachten betreft overeenstemmen met het type van de actuele parameter. Een variabele van het type Tabell stemt echter niet overeen met het type Tabel2. Dit impliceert dat we voor beide typen een afzonderlijke procedure zullen moeten declareren.
Voor talrijke toepassingen is deze strikte typekoppeling van tabellen te streng. De 'open array'-parameter biedt hier een oplossing. Een open-array-parameter legt vast dat de formele parameter wordt gekoppeld aan een actuele parameter met de volgende kenmerken :
- de actuele parameter is een eendimensionale tabel;
- het elementtype van de actuele parameter is gelijk aan het elementtype van de formele parameter;
- het indextype van de actuele parameter is niet bekend.

Met een open-array-parameter worden tabellen met een willekeurig aantal elementen aan een formele parameter gekoppeld. De syntaxis voor een open-array-parameter is :

```
FormeelType = [ 'ARRAY' 'OF' ] kwalificatie
```

In de procedurerromp kennen we het echte indextype van de actuele parameter niet. We kunnen elk element van de tabel adresseren als we rekening houden met de eigenschappen :
- de ondergrens van de index van een open-array-parameter is steeds nul;
- de bovengrens van de index van een open-array-parameter a wordt geleverd door de standaardfunctie HIGH(a);
- het aantal elementen van een open-array-parameter a is gelijk aan HIGH(a) + 1.

Voorbeeld :

```
VAR a : ARRAY [1..10] OF CARDINAL;
    b : ARRAY [1..1000] OF CARDINAL;
    c : ARRAY ['a'..'z'] OF CARDINAL;
    d : ARRAY (maandag, dinsdag, woensdag, donderdag,
               vrijdag) OF CARDINAL;
```

De procedure DrukTabel drukt de elementen van een tabel na elkaar af :

```
PROCEDURE DrukTabel(tabel : ARRAY OF CARDINAL);
VAR i : CARDINAL;

BEGIN
FOR i := 0 TO HIGH(tabel) DO
  WriteCard(tabel[i],8);
  WriteLn
END

END DrukTabel;
```

Met deze procedure kunnen de waarden van de tabellen a, b, c of d worden afgedrukt.

Opmerking : In Modula-2 is het gebruik van open-array-parameters beperkt tot eendimensionale tabellen.

6.4 Het bereik van objecten

Een procedure heeft dezelfde vorm als een programmamodule. In een procedure kunnen we dezelfde soorten objecten declareren als in een programma : constanten, typen, variabelen en procedures. De objecten die we in een procedure declareren zijn lokaal. De procedure bakent het tekstdeel in het programma af waar de identifiers van deze objecten een betekenis hebben, met andere woorden de procedure definieert het bereik (Engels : scope) van de identifier. De regels voor het bereik van een identifier zijn :

- het bereik van een identifier is de procedure waarin de identifier is gedeclareerd. Indien de procedure andere proceduredeclaraties bevat, behoren ook deze procedures tot het bereik van de identifier, tenzij de volgende regel geldt;
- een identifier i wordt gedeclareerd in procedure P. De procedure Q wordt binnen P gedeclareerd. Als nu in procedure Q opnieuw een identifier i wordt gedeclareerd, vallen Q en alle procedures binnen Q buiten het bereik van de in P gedefinieerde i;
- de Modula-2 standaardidentifiers behouden hun betekenis in de volledige programmatekst. Deze identifiers zijn als het ware gedefinieerd in een procedure die het gehele programma omsluit.

Bij elke aanroep van een procedure zijn de waarden van de lokale variabelen niet gedefinieerd. De waarde van een lokale variabele gaat verloren bij het beëindigen van de procedure waar de variabele is gedefinieerd. De levensduur van de lokale variabelen is dus beperkt tot de tijd dat de procedure actief is.

Voorbeeld :

```
                                bereik

                             iP1  j  iP3

PROCEDURE P1;
VAR i : CARDINAL;             |
...                           |
  PROCEDURE P2;               |
  VAR j : CARDINAL;           |   |
  ...                         |   |
  END P2;                     |   |
                              |
  PROCEDURE P3;               |
  VAR i : CHAR;                       |
  ...                                 |
  END P3;                             |
...                           |
END P1;                       |
```

De variabele i wordt gedeclareerd in de procedure P1. Voor procedure P2 is i een globale variabele : een variabele die niet in de procedure zelf is gedefinieerd. De variabele j wordt in de procedure P2 gedeclareerd. Het bereik van j is beperkt tot de procedure P2. In de procedure P3 wordt opnieuw een variabele i gedeclareerd. Het bereik van deze variabele is beperkt tot de procedure P3. Door deze declaratie wordt het bereik van de variabele i, gedeclareerd in procedure P1 beperkt; het bereik van P3 valt er nu buiten.

In principe vermijden we het gebruik van globale variabelen : de leesbaarheid van de procedures wordt erdoor geschaad en de procedures zijn niet algemeen opnieuw te gebruiken. In sommige situaties is het gebruik van globale variabelen nog geoorloofd, bijvoorbeeld voor het uitwisselen van informatie tussen twee of meer procedures of voor het bewaren van de waarde van een variabele tussen opeenvolgende procedure-aanroepen.

Samenvatting syntaxis

```
ProcedureDeclaratie = ProcedureKop ';' blok identifier
ProcedureKop        = 'PROCEDURE' identifier [ FormeleParameters ]
```

```
blok                = { declaratie }
                      [ 'BEGIN' OpdrachtenRij ]
                      'END'
FormeleParameters   = '(' [FPSectie {';' FPSectie } ] ')'
FPSectie            = [ 'VAR' ] IdentLijst ':' FormeelType
FormeelType         = [ 'ARRAY' 'OF' ] kwalificatie

ProcedureAanroep    = aanwijzing [ ActueleParameters ]
ActueleParameters   = '(' [ UitdLijst ] ')'
```

We kunnen nu ook de syntaxis voor de opdrachten voor de declaraties verder aanvullen :

```
declaratie = 'CONST' { ConstDeclaratie ';' }
           | 'TYPE' { TypeDeclaratie ';' }
           | 'VAR'  { VariabeleDeclaratie ';' }
           | ProcedureDeclaratie ';'
opdracht =   [ ToekenningsOpdracht
           | ProcedureAanroep
           | WhileOpdracht
           | RepeatOpdracht
           | ForOpdracht
           | LoopOpdracht
           | IfOpdracht
           | CaseOpdracht
           | WithOpdracht
           | ReturnOpdracht
           | 'EXIT' ]
```

6.5 De functieprocedure

In talrijke situaties levert een procedure als uitvoer slechts één waarde. Dit resultaat wordt dikwijls direct na de aanroep in een uitdrukking gebruikt. We definiëren bijvoorbeeld de procedure BerekenGemiddelde voor het berekenen van het rekenkundig gemiddelde van de elementen van een tabel. Het aantal elementen is willekeurig, maar steeds groter dan nul.

```
PROCEDURE BerekenGemiddelde(    a : ARRAY OF CARDINAL;
                            VAR g : CARDINAL);
VAR i   : CARDINAL;
    som : CARDINAL;

BEGIN
som := 0;
```

```
FOR i := Ø TO HIGH(a) DO
  INC(som,a[i])
END;
g := som DIV (HIGH(a) + 1)

END BerekenGemiddelde;
```

Deze procedure geeft als enig resultaat de waarde g, het gemiddelde. Als we in een programma het rekenkundig gemiddelde van een tabel nodig hebben, coderen we eerst een aanroep van de procedure BerekenGemiddelde en gebruiken vervolgens de actuele parameter in een uitdrukking, bijvoorbeeld :

```
BerekenGemiddelde(a,gemiddelde);
WriteCard(gemiddelde,5)
```

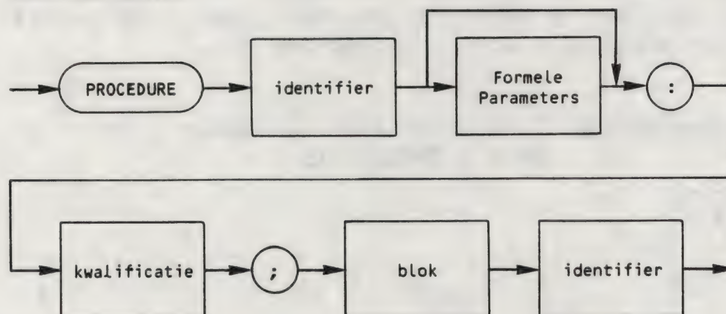
Een functieprocedure gebruikt het resultaat van een functie direct in een uitdrukking. Het type van het resultaat is een scalair of een samengesteld type. Voor sommige Modula-2 implementaties zijn echter alleen scalaire functies toegelaten.

6.6 De declaratie van een functieprocedure

De verschillen met de declaratie van een gewone procedure zijn :

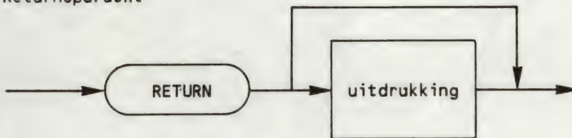
- de kopregel van de functieprocedure vermeldt het resultaattype van de functie na de formele parameterlijst. Hiervoor geldt de syntaxisregel :

```
FormeleParameters =
     '(' [ FPSectie { ';' FPSectie } ] ')' ':' kwalificatie
```



- de kopregel bevat altijd een parameterlijst. Deze lijst is eventueel leeg en wordt dan voorgesteld door '()';
- het opdrachtendeel bevat ten minste een RETURN-opdracht gevolgd door een uitdrukking. De waarde van deze uitdrukking is het resultaat van de functie. De syntaxis voor de RETURN-opdracht is :

ReturnOpdracht = 'RETURN' uitdrukking



Het resultaattype van een functie is een scalair of een samengesteld type. Bij sommige implementaties worden uitsluitend scalaire functies toegelaten. Voor de functienaam gebruiken we bij voorkeur een zelfstandig naamwoord; voor de namen van de functies van het type BOOLEAN kiezen we een adjectief of een omschrijving.

Voorbeelden :

We definiëren het berekenen van het gemiddelde nog eens, maar nu als een functieprocedure. Het resultaattype van de procedure is CARDINAL :

```
PROCEDURE Gemiddelde(a : ARRAY OF CARDINAL) : CARDINAL;
VAR i   : CARDINAL;
    som : CARDINAL;

BEGIN
som := 0;
FOR i := 0 TO HIGH(a) DO
  INC(som,a[i])
END;
RETURN som DIV (HIGH(a) + 1)

END Gemiddelde;
```

Met de procedure Aanwezig wordt een element opgezocht in een tabel. Als het element wordt gevonden is het resultaat TRUE, anders FALSE. Het resultaattype van de procedure is dus BOOLEAN.

```
PROCEDURE Aanwezig(    a       : ARRAY OF CHAR;
                       element : CHAR) : BOOLEAN;
VAR i : CARDINAL;

BEGIN
FOR i := Ø TO HIGH(a) DO
  IF a[i] = element
  THEN
    RETURN TRUE
  END
END;
RETURN FALSE

END Aanwezig;
```

Het volgende voorbeeld is een procedure voor het berekenen van de lengte van een rij tekens of een string. De definitie van een string in Modula-2 is

```
CONST maxLengte = ...; (* afhankelijk van de toepassing *)
TYPE  String    = ARRAY [Ø..maxLengte - 1] OF CHAR;
```

De waarde van een string bestaat uit Ø tot maxLengte tekens. Als de string minder dan maxLengte tekens bevat, wordt de rij afgesloten met het teken ØC. De declaratie van de functieprocedure voor het berekenen van de lengte van een string is dan :

```
PROCEDURE Length(s : ARRAY OF CHAR) : CARDINAL;
VAR i : CARDINAL;

BEGIN
i := Ø;
WHILE (i <= HIGH(s)) AND (s[i] # ØC) DO
  INC(i)
END;
RETURN i
END Length;
```

6.7 De functie-aanroep

Een functieprocedure wordt direct in een uitdrukking aangeroepen met een functie-aanroep. De syntaxis is dezelfde als voor de procedure-aanroep. De actuele parameterlijst is verplicht maar kan eventueel leeg zijn.

Voorbeelden :

```
a := Gemiddelde(tabel) + 5;
IF Aanwezig(s,teken) THEN ...
```

6.8 Neveneffecten

In een functieprocedure kunnen we ook een waarde toekennen aan variabelen die buiten de functie zijn gedefinieerd. Slechts èèn waarde wordt echter als resultaat van de functie in een uitdrukking gebruikt. De andere variabelen worden doorgegeven met variabele-parameters of door het toekennen van een waarde aan een globale variabele door middel van een toekenningsopdracht. Het veranderen van de waarde van deze variabelen is een neveneffect van de functie.

Voorbeeld :

```
PROCEDURE TekenAanwezig(VAR t : CHAR) : BOOLEAN;

BEGIN
ReadChar(t);
RETURN NOT EOL()

END TekenAanwezig;
```

De procedure TekenAanwezig onderzoekt de aanwezigheid van een teken in de standaard-invoerstroom. Het resultaat van de functie is

- TRUE als een teken is gelezen of FALSE als het teken 'einde regel' is ingevoerd;
- de waarde van t is het gelezen teken, als het resultaat TRUE is; de waarde van t is niet gedefinieerd, als het resultaat FALSE is.

```
PROCEDURE Kwadraat(x : CARDINAL) : CARDINAL;
BEGIN
INC(teller);
RETURN x * x

END Kwadraat;
```

Deze procedure berekent het kwadraat van de parameter. De variabele teller telt het aantal keer dat de functie wordt aangeroepen. Als we echter de variabele teller en de functie-aanroep in dezelfde rekenkundige uitdrukking gebruiken, kan de commutatieve eigenschap van 'het optellen' verloren gaan.

Kwadraat(m) + teller # teller + Kwadraat(m)

Dit komt omdat als neveneffect van de functie Kwadraat de waarde van de uitdrukkingen

Kwadraat(m) + teller

en

teller + Kwadraat(m)

verschillend is. We vermijden daarom het gebruik van functies met neveneffecten en we vervangen ze door een gewone procedure met variabele-parameters. Slechts in uitzonderlijke, goed beheersbare situaties is het gebruik van een functie met neveneffecten geoorloofd.

6.9 Het proceduretype en de procedurevariabele

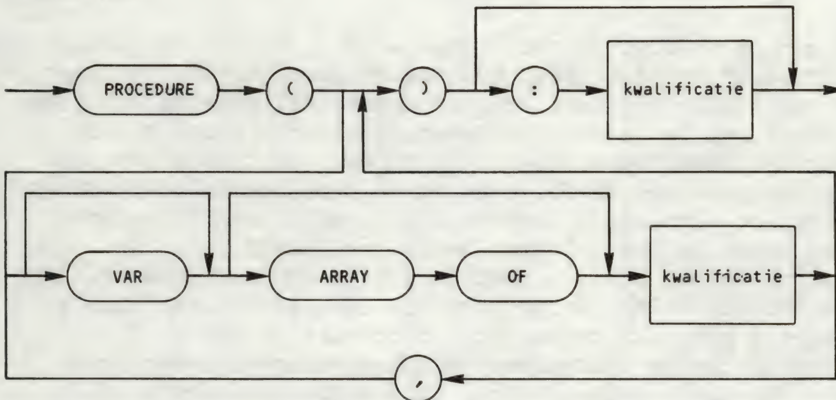
Een procedure is een deel van het programma waarin de bewerkingen op een aantal variabelen worden gespecificeerd. We kunnen een procedure ook beschouwen als een programma-object, net zoals constanten, typen en variabelen. De procedure-declaratie en een constantedeclaratie zijn gelijksoortig. Een procedurevariabele is een variabele waarvoor de waarde een procedure is. Een proceduretype definieert de verzameling waarden die aan een procedurevariabele kunnen worden toegekend. Een proceduretype definieert het aantal parameters, het type en de koppeling van de parameters. Voor een functieprocedure wordt ook het resultaattype aangegeven.

Voorbeelden :

```
TYPE RealFunc = PROCEDURE (REAL) : REAL;
     ProcCard = PROCEDURE (VAR CARDINAL, CARDINAL);
```

Het type RealFunc definieert de verzameling procedurewaarden met een REAL waardeparameter en een REAL functiewaarde. Enkele waarden voor dit type zijn Sin, Cos, Ln. Het type ProcCard definieert de verzameling procedurewaarden met een variabele-parameter en een waardeparameter met type CARDINAL. Een waarde voor dit type is WriteCard.
De syntaxis voor de declaratie van een proceduretype is :

```
ProcedureType    = 'PROCEDURE' [ FormeleTypeLijst ]
FormeleTypeLijst =
 '(' [ [ 'VAR' ] FormeelType { ',' [ 'VAR' ] FormeelType } ] ')'
 [ ':' identifier ]
```



Het standaardtype PROC definieert de procedures zonder een parameterlijst :

```
PROC = PROCEDURE;
```

We definiëren de variabelen

```
VAR f, g : RealFunc;
    p    : ProcCard;
```

Voor de procedurevariabelen zijn alleen de toekenningsopdracht en de vergelijking gedefinieerd :

```
f := Sin;
g := Ln;
p := ReadCard;
IF f # g THEN .. END
```

De opdracht

```
p(a,10)
```

is nu equivalent met de opdracht

```
ReadCard(a,10)
```

6.10 Toepassingen van procedurevariabelen

Met een proceduretype declareren we de parameterstructuur van een procedure of een functie. Een waarde van dit type gedraagt zich als een verwijzing naar een procedure. Elke verwijzing naar een procedure met de juiste parameterstructuur kunnen we als waarde aan de procedurevariabele toekennen. De plaats van deze waarde in het geheugen is willekeurig en procedurewaarden kunnen we eventueel ook dynamisch creëren. De creatie van dynamische variabelen wordt in het volgende hoofdstuk beschreven. De proceduretypen mogen ook een onderdeel zijn van een datastructuur zoals een RECORD, ARRAY of een bestand. We illustreren enkele toepassingen : procedureparameters, procedurevariabelen in bibliotheekmodulen en procedures voor het behandelen van datastructuren die in de datastructuur zelf worden gedefinieerd.

Procedureparameters

Met het proceduretype kunnen we procedures definiëren waarbij procedures als parameter in de formele parameterlijst voorkomen. Bij de activering van de procedure wordt de actuele procedurewaarde aan de formele parameter toegekend.

Voorbeeld :

Procedureparameters worden zeer dikwijls gebruikt in wiskundige toepassingen.

```
TYPE RealFunc = PROCEDURE(REAL) : REAL;
```

```
PROCEDURE Integreer(f          : RealFunc;
                    ondergrens : REAL;
                    bovengrens : REAL) : REAL;


PROCEDURE Plot     (f          : RealFunc;
                    ondergrens : REAL;
                    bovengrens : REAL);
```

Voor de functie **sinus** kunnen we bijvoorbeeld de procedure als volgt aanroepen :

```
y := Integreer(Sin,-0.1,0.1);
```

Procedurevariabelen in bibliotheekmodulen

Met Modula-2 bibliotheekmodulen kunnen we programmatuur ontwikkelen die algemeen te gebruiken is. De programmatuurcomponenten die aldus ontstaan zijn echter niet steeds volledig aangepast aan de behoeften van diegenen die de modulen gebruiken. Zo bevatten bijvoorbeeld modulen dikwijls procedures voor het afhandelen van fouten. De ene gebruiker zal deze ingebouwde procedures wel willen gebruiken, terwijl andere gebruikers deze procedures willen vervangen door andere procedures voor de foutenafhandeling. Met procedurevariabelen kunnen we dergelijke problemen oplossen : aan procedurevariabelen kennen we de waarde toe van de gewenste procedures. Het effect van de aanroep van een procedurevariabele is de aanroep van de procedure die als waarde aan de variabele is toegekend. We illusteren deze techniek met een eenvoudig voorbeeld. We gebruiken hiervoor de standaardmodule Directory voor het beheer van de inhoudstabellen van externe geheugens. In deze module worden onder meer de volgende objecten gedefinieerd :

```
TYPE DirQueryProc = PROCEDURE(ARRAY OF CHAR, VAR BOOLEAN);
PROCEDURE DirQuery(    wildName  : ARRAY OF CHAR;
                       dirProc   : DirQueryProc;
                   VAR status    : FileState);
```

De invoerparameter wildName is een dubbelzinnige of ondubbelzinnige bestandsidentificatie. De syntaxis van de identificatie is afhankelijk van het systeem. Voor een personal computer is een bestandsidentificatie bijvoorbeeld

A*.REF

De procedure DirQuery herhaalt de aanroep van de procedure dirProc voor elke bestandsidentificatie die aan de invoerparameter voldoet en stopt indien geen bestandsidentificaties meer worden gevonden of indien de procedure dirProc een FALSE-waarde geeft. Met de procedure DirQuery ondervragen we de inhoudstabel of manipuleren we een verzameling bestanden waarnaar wordt verwezen. De eigenlijke verwerking wordt bepaald door de procedure dirProc.

We definiëren een procedure DrukBestandsnaam voor het afdrukken van een bestandsnaam.

```
PROCEDURE DrukBestandsnaam(    naam      : ARRAY OF CHAR;
                           VAR resultaat : BOOLEAN);
```

Aan de parameter naam wordt de waarde toegekend die moet worden afgedrukt. De waarde van de resultaat parameter is afhankelijk van het resultaat van de verwerking en wordt bijvoorbeeld interactief bepaald. We gebruiken in DrukBestandsnaam de bewerkingen CondRead, WriteString en WriteLn van de module SimpleIO.

```
PROCEDURE DrukBestandsnaam(    naam      : ARRAY OF CHAR;
                           VAR resultaat : BOOLEAN);
VAR ok    : BOOLEAN;
    teken : CHAR;

BEGIN
WriteString(naam);
WriteLn;
CondRead(teken,ok);
resultaat := NOT ok OR (CAP(teken) # 'S')
END DrukBestandsnaam;
```

Met de volgende declaraties en opdrachten worden de bestandsnamen afgedrukt die voldoen aan de bestandidentificatie **A*.REF**. We selecteren deze bestandsnamen met de procedure DirQuery.

```
FROM Files     IMPORT FileState;
FROM Directory IMPORT DirQuery;

VAR status : FileState;


DirQuery('A*.REF', DrukBestandsnaam, status);
```

Het werken met een datastructuur

Als laatste voorbeeld beschrijven we een toepassing van procedurevariabelen voor het werken met datastructuren. De procedure voor de bewerking is zelf een onderdeel van de datastructuur. We beschouwen de volgende declaraties :

```
TYPE Gegevens = RECORD
       (* typering van de te bewerken gegevens *)
     END;

TYPE Bewerking = PROCEDURE (REAL, INTEGER, VAR Gegevens);
     Element     = RECORD
       bewerking : Bewerking;
       gegevens    : Gegevens
     END;
```

De datastructuur Element is samengesteld uit twee velden : een procedurebewerking en een 'gegevens'-structuur waarop de bewerking moet worden uitgevoerd. Voor elk gegevenselement kunnen we de passende bewerking in de structuur invullen.

```
PROCEDURE Transactiel(    x : REAL;
                          y : INTEGER;
                      VAR g : Gegevens);

BEGIN
...
END Transactiel;


PROCEDURE Transactie2(    x : REAL;
                          y : INTEGER;
                      VAR g : Gegevens);

BEGIN
...
END Transactie2;

VAR a,b : Element;

BEGIN
...
a.bewerking := Transactiel;
b.bewerking := Transactie2;
```

Literatuur

Mc Cormack, Gleaves, 'Modula-2, a Worthy Successor to Pascal, Proceduretypes',
Byte, april 1983

Coar D., 'Pascal, ADA and Modula-2, A System Programmer's Comparision',
Byte, augustus 1984

Sumner, R., 'Dynamic Module Instantiation',
Modula-2 News, april 1985

Oefeningen

1. Beschouw de volgende declaraties :

```
CONST b = 2;
VAR   x : CARDINAL;
      y : CARDINAL;
      z : CARDINAL;
      w : REAL;

PROCEDURE A(    i : CARDINAL;
                y : CARDINAL;
            VAR j : CARDINAL);
BEGIN
i := 5;
j := i + y;
INC(y)
END A;
...
x := 4;
y := 0;
z := 3;

(* aanroep procedure A *)
```

Bepaal het resultaat van de volgende aanroepen van procedure A. Veronderstel elke aanroep onmiddellijk na de opdracht z := 3. Zoek ook eventuele foute aanroepen.

a. A(x, y, z)
b. A(b, y, z)
c. A(x, y, b)
d. A(z, y, z)
e. A(y, y, b)

f. A(x, y, x + z)
g. A(x + y, y, z)
h. A(x, y, w)

2. Schrijf een procedure voor het ordenen van drie getallen met type REAL in niet-dalende volgorde. Test de procedure met alle mogelijke permutaties van de getallen -1.Ø, 2.Ø en 3.Ø.

3. Schrijf een procedure Sorteer voor het sorteren van een tabel van klein naar groot. Het elementtype is CARDINAL. Gebruik de tussenvoegmethode en open-array-parameters.

4. Schrijf een procedure Meng voor het mengen van twee van klein naar groot gesorteerde tabellen tot een nieuwe tabel. Het basistype van de elementen is CARDINAL. Gebruik open-array-parameters.

5. Gegeven de lengten a, b en c van de zijden van een driehoek. Schrijf een functieprocedure met resultaattype BOOLEAN om na te gaan dat a, b en c werkelijk de zijden van een driehoek voorstellen. Het type van de zijden is REAL.

6. Schrijf een functieprocedure Faculteit voor het berekenen van n!. Het type van Faculteit en n is CARDINAL.

7. Een Romeins getal is afhankelijk van de volgende letter-cijfer conversies :

M = 1ØØØ
D = 5ØØ
C = 1ØØ
L = 5Ø
X = 1Ø
V = 5
I = 1

De waarde van een Romeins getal is de som van de waarden van de Romeinse cijfers :

III = 3
MMII = 2ØØ2
MCLXVII = 1167

De volgorde van de Romeinse cijfers in een getal is steeds dalend. De cijfers D, L en V mogen slechts een keer in een getal voorkomen. De overige cijfers mogen vier keer na elkaar worden herhaald. Met deze definitie kunnen we de getallen tot 4999 voorstellen.
Schrijf de procedures voor de omzetting van een Romeins getal naar een decimaal getal en omgekeerd. Een Romeins getal wordt voorgesteld met het type :

```
TYPE RomeinsGetal = ARRAY [Ø..2Ø] OF CHAR;
```

8. Soms worden de Romeinse cijfers C, X en I een positie uit volgorde geschreven om een aftrekking aan te geven. Bijvoorbeeld :

IV = 4
IX = 9
XL = 4Ø
XC = 9Ø
CD = 4ØØ
CM = 9ØØ

Wijzig de procedures van opgave 7 voor deze nieuwe notatie.

9. Schrijf een functieprocedure ScalairProdukt voor het berekenen van het scalair produkt van twee vectoren met elementtype INTEGER. Het scalair produkt voor twee vectoren a en b wordt gedefinieerd als :

ScalairProdukt = $a_1 * b_1 + a_2 * b_2 + a_3 * b_3 + \ldots + a_n * b_n$

Gebruik open-array-parameters.

1Ø. Schrijf een procedure KeerOm voor het omkeren van een string tekens. Het argumenttype is :

```
TYPE S = ARRAY [Ø..2Ø] OF CHAR;
```

Gebruik slechts één hulpvariabele met type CHAR.

11. Schrijf een procedure WisHerhaling die in een string tekens nagaat of een teken meer dan eens na elkaar voorkomt. Wis alle herhalingen.

7 Recursie

7.1 Iteratie en recursie

Met een iteratieve herhalingsstructuur specificeren we de opdrachtenrij die meer dan een keer moet worden uitgevoerd en de conditie waaronder dit moet gebeuren. Met deze herhalingsstructuur moet elke iteratie, de totale uitvoering van de opdrachtenrij, steeds volledig worden afgewerkt voordat een nieuwe iteratie begint. Een iteratieve herhalingsstructuur noemen we meestal een lus, de opdrachtenrij is de romp van de lus. De opdrachten WHILE DO, REPEAT UNTIL en LOOP zijn iteratieve herhalingsstructuren.

Een recursieve herhalingsstructuur is algemener dan de iteratie : de opdrachtenrij in de romp hoeft niet volledig te worden afgewerkt voordat aan een nieuwe herhaling wordt begonnen. Met een recursieve herhalingsstructuur kan de tweede herhaling worden gestart voordat de eerste volledig is afgewerkt. Als we later een herhaling verder willen afwerken, moeten we weer kunnen beschikken over de toestandsruimte van het ogenblik dat de herhaling is onderbroken. Deze situatie is analoog met de aanroep van een subprogramma : na de beëindiging van het subprogramma wordt het aanroepende programma verder verwerkt met de toestandsruimte die geldig was op het ogenblik van de aanroep, het effect van het subprogramma op de toestandsruimte niet inbegrepen. De recursieve herhaling wordt ook in Modula-2 geïmplementeerd door een procedure die zichzelf aanroept. Zoals bij de iteratieve herhalingsstructuur zorgen we ook in een recursieve procedure voor een besturingsstructuur, zodat de herhaling kan worden gestopt. In een recursieve procedure coderen we de aanroep als één van de alternatieven van een selectie-opdracht (IF- of CASE-opdracht) zodat in een bepaald geval de procedure wordt aangeroepen en in andere gevallen niet. De iteratie

```
WHILE conditie DO s END
```

kunnen we als volgt recursief formuleren

```
TYPE Verwerking = PROC;
```

```
PROCEDURE WhileDo(conditie : BOOLEAN;
                  s        : Verwerking);
BEGIN
IF conditie
THEN
  s;
  WhileDo(conditie,s)
END
END WhileDo;
```

Op een zelfde wijze formuleren we de iteratie

REPEAT s UNTIL conditie

```
PROCEDURE RepeatUntil(s        : Verwerking;
                      conditie : BOOLEAN);
BEGIN
s;
IF NOT conditie
THEN
  RepeatUntil(s,conditie)
END
END RepeatUntil;
```

Bij elke aanroep van een recursieve procedure wordt een nieuwe toestandsruimte voor de procedure gecreëerd. De implementatie van een herhalingsstructuur met behulp van recursie geeft snel aanleiding tot een tekort aan geheugenruimte en is daardoor over het algemeen minder efficiënt dan een iteratieve herhalingsstructuur. Recursieve algoritmen die we ook met een iteratiestructuur kunnen formuleren zijn pseudo-recursief en kunnen beter daardoor worden vervangen.

Voorbeelden :
Binair zoeken

We gebruiken de declaraties :

```
CONST n         = 100;
TYPE  Index     = [0..n + 1];
      Tabel     = ARRAY Index OF CARDINAL;
      IndexNul  = [0..n];
```

We maken een functie PlaatsBinair. Deze functie evalueert de plaats van een element in een geordende tabel met type Tabel. Als het element in de tabel voorkomt behoort het resultaat van de functie tot het interval [1..n]; als het element niet in de tabel voorkomt is het resultaat nul.

Het algoritme voor binair zoeken is reeds beschreven in het hoofdstuk 'Samengestelde typen'. De recursieve oplossing steunt op de grootte van een deeltabel die nog moet worden onderzocht. De deeltabel wordt bepaald door de indicen laag en hoog. Als deze tabel leeg is, dit is als laag groter is dan hoog, komt het element niet voor in de tabel en is de functiewaarde nul. Bij elke aanroep bepalen we het middelste element van de tabel. Als dit middelste element gelijk is aan het op te zoeken element is de functiewaarde de index van het middelste element en stopt het algoritme. In het andere geval onderzoeken we een van de volgende deeltabellen :

(1) de onderste helft van de tabel van laag tot midden - 1;
(2) de bovenste helft van de tabel van midden + 1 tot hoog.

Het zoeken in de tabel wordt dus bij elke herhaling herleid tot een eenvoudiger zoekopdracht in de onderste of bovenste helft van het nog te onderzoeken deel van de tabel. De herhaling stopt bij het vinden van het gezochte element of bij het zoeken in een lege deeltabel.

```
PROCEDURE PlaatsBinair(VAR a       : Tabel;
                           element : CARDINAL;
                           laag    : Index;
                           hoog    : Index) : IndexNul;
VAR midden : Index;

BEGIN
IF laag <= hoog
THEN
  midden := (laag + hoog) DIV 2;
  IF element = a[midden]
  THEN
    RETURN midden
  ELSIF element < a[midden]
  THEN
    RETURN PlaatsBinair(a, element, laag, midden - 1)
  ELSE
    RETURN PlaatsBinair(a, element, midden + 1, hoog)
  END
ELSE
  RETURN 0
END
END PlaatsBinair;
```

De functieprocedure PlaatsBinair heeft twee belangrijke eigenschappen : de twee recursieve aanroepen komen elk voor in een tak

van een IF-opdracht, zodat steeds slechts een van beide tijdens het uitvoeren van de lus wordt verwerkt. Ook bevat de procedure na de recursieve aanroepen geen opdrachten meer. Dit betekent dat er slechts aan een nieuwe lus wordt begonnen indien de huidige opdrachtenrij volledig is afgewerkt. Deze eigenschap geldt ook voor een iteratieve herhalingsstructuur. We kunnen voor PlaatsBinair ook een oplossing formuleren met een iteratie :

```
PROCEDURE PlaatsBinair(VAR a       : Tabel;
                           element : CARDINAL;
                           laag    : Index;
                           hoog    : Index) : IndexNul;
VAR midden : Index;
BEGIN
WHILE laag <= hoog DO
  midden := (laag + hoog) DIV 2;
  IF element = a[midden]
  THEN
    RETURN midden
  ELSIF element < a[midden]
  THEN
    hoog := midden - 1
  ELSE
    laag := midden + 1
  END
END;
RETURN Ø
END PlaatsBinair;
```

Veel problemen worden recursief gedefinieerd. De algoritmen voor de oplossing van deze problemen worden eenvoudig vertaald in recursieve procedures. We illustreren dit met enkele voorbeelden : De faculteit van een natuurlijk getal wordt gedefinieerd als

Ø! = 1
n! = n * (n - 1)! voor n > Ø

```
PROCEDURE Faculteit(n : CARDINAL) : CARDINAL;
BEGIN
IF n <= 1
THEN
  RETURN 1
ELSE
  RETURN n * Faculteit(n - 1)
END
END Faculteit;
```

Het berekenen van de som van de cijfers in een getal. Deze som kunnen we als volgt definiëren :

```
SomCijfers(getal) = getal
                    voor getal < 10
SomCijfers(getal) = getal MOD 10 + SomCijfers(getal DIV 10)
                    voor getal >= 10
```

```
PROCEDURE SomCijfers(getal : CARDINAL) : CARDINAL;
BEGIN
IF getal < 10
THEN
  RETURN getal
ELSE
  RETURN getal MOD 10 + SomCijfers(getal DIV 10)
END
END SomCijfers;
```

De recursieve aanroep in deze beide voorbeelden is telkens de laatste opdracht van de lus. Een oplossing met een iteratie-structuur is dus ook mogelijk.
We formuleren nu de procedure SomCijfers met twee lokale variabelen, eenheden en tientallen.

```
PROCEDURE SomCijfers(getal : CARDINAL) : CARDINAL;
VAR eenheden, tientallen : CARDINAL;

BEGIN
IF getal < 10
THEN
  RETURN getal
ELSE
  eenheden := getal MOD 10;
  tientallen := getal DIV 10;
  RETURN eenheden + SomCijfers(tientallen)
END
END SomCijfers;
```

Iedere recursieve aanroep van de procedure creëert de lokale variabelen eenheden en tientallen terwijl de variabelen van de vorige aanroep nog blijven bestaan. De variabelen zijn volledig onafhankelijk van elkaar hoewel ze dezelfde naam hebben. De levensduur van elke verzameling lokale variabelen is steeds gelijk aan de tijd dat de procedure actief is. Voor een verzameling lokale variabelen die in een recursieve procedure R is gedefinieerd, genereert de procedure R na k recursieve aanroepen dus k + 1

verzamelingen lokale variabelen. Deze eigenschappen gebruiken we in het volgende voorbeeld : de procedure Keerom leest een reeks tekens van de standaardinvoer en drukt deze reeks omgekeerd op de standaarduitvoer af :

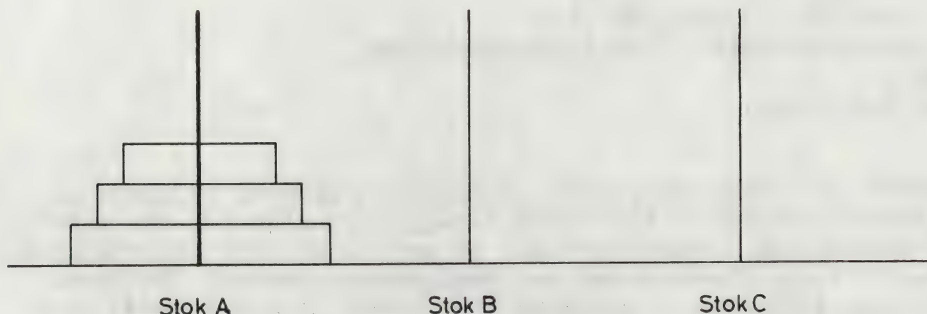
```
PROCEDURE Keerom;
VAR c : CHAR;

BEGIN
IF NOT EOL()
THEN
  ReadChar(c);
  Keerom;
  WriteChar(c)
END
END Keerom;
```

We beëindigen dit hoofdstuk met het klassieke 'Torens van Hanoi'-probleem en enkele daarvan afgeleide varianten.

De torens van Hanoi

Stel dat we beschikken over een plank met drie stokken A, B en C. Over stok A zijn n schijven, alle met een verschillende diameter, zo geplaatst dat geen enkele schijf op een kleinere schijf rust. We ontwikkelen een procedure om de schijven van A naar C te verplaatsen. Alleen de bovenste schijf van een stok mag naar een andere worden verplaatst en een grotere schijf mag nooit op een kleinere schijf rusten.



figuur 7.1 De Torens van Hanoi

We veronderstellen dat we het probleem kunnen oplossen voor n-1 schijven. Als we een algoritme kunnen formuleren op grond van n-1 schijven, beschikken we over een recursief algoritme :
(1) als n = 1, verplaats dan de schijf van A naar C; stop de verwerking;
(2) verplaats de n-1 bovenste schijven van A naar B met C als hulpstok;
(3) verplaats de overblijvende schijf van A naar C;
(4) verplaats de n-1 schijven van B naar C met A als hulpstok.
In de oplossing geven we de verplaatsing van een schijf weer met de volgende tekst :
'verplaats schijf <volgnummer> van stok <oorsprong> naar stok <bestemming>'
We nummeren de schijven van boven naar beneden : de bovenste schijf krijgt volgnummer 1, de onderste volgnummer n. We gebruiken hiervoor de bewerkingen WriteString, WriteChar en WriteLn van de module SimpleIO. Het verplaatsen van de schijven kunnen we ook op het scherm weergeven zodat de uitvoer van de procedure als een tekenfilm wordt getoond.

```
TYPE Stok = ['A'..'C'];
PROCEDURE VerwerkTorensHanoi(    aantal : CARDINAL;
                                 van    : Stok;
                                 naar   : Stok;
                                 hulp   : Stok);

  PROCEDURE Verplaats(    volgnummer : CARDINAL;
                          van        : Stok;
                          naar       : Stok);
  BEGIN
    WriteString('verplaats schijf');
    WriteCard(volgnummer, 2);
    WriteString(' van stok ');
    WriteChar(van);
    WriteString(' naar stok ');
    WriteChar(naar);
    WriteLn
  END Verplaats;

BEGIN
IF aantal = 1
THEN
  Verplaats(1, van, naar)
ELSE
  VerwerkTorensHanoi(aantal - 1, van, hulp, naar);
  Verplaats(aantal, van, naar);
  VerwerkTorensHanoi(aantal - 1, hulp, naar, van)
END
END VerwerkTorensHanoi;
```

Met de aanroep

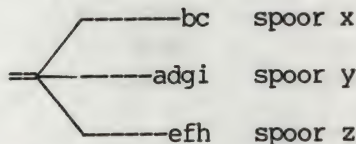
```
VerwerkTorensHanoi(3,'A','B','C')
```

wordt het volgende resultaat afgedrukt :

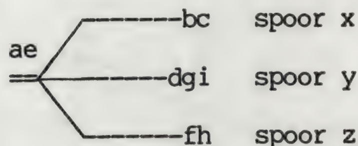
```
verplaats schijf 1 van stok A naar stok B
verplaats schijf 2 van stok A naar stok C
verplaats schijf 1 van stok B naar stok C
verplaats schijf 3 van stok A naar stok B
verplaats schijf 1 van stok C naar stok A
verplaats schijf 2 van stok C naar stok B
verplaats schijf 1 van stok A naar stok B
```

Het spoorwegprobleem

Een variant van de Torens van Hanoi is het volgende spoorwegprobleem :
een aantal wagons bevindt zich op verschillende sporen. De wagons worden aangeduid met de letters 'a' tot en met 'i', de sporen met de letters 'x', 'y' en 'z'. Elke rij wagons op een spoor moet op alfabetische volgorde opgesteld staan. We kunnen de situatie op een gegeven ogenblik als volgt voorstellen :



Een gemeenschappelijk spoor verbindt de sporen 'x', 'y' en 'z'. Dit spoor biedt plaats voor de locomotief ('a') en één wagon.



Gevraagd wordt een procedure om de trein op een gegeven spoor samen te stellen. De invoer van het programma is de startconfiguratie en de bestemming. De uitvoer is in een eerste oplossing de opsomming van de uitgevoerde verplaatsingen, in een tweede oplossing een schematische weergave van de actuele configuratie op elk ogenblik van de verwerking.

De algemene oplossing van het probleem bestaat uit de volgende drie stappen :
(1) lees de startconfiguratie;
(2) lees de bestemming;
(3) formeer de trein.

Voor de oplossing gebruiken we de volgende structuren en procedures :

```
TYPE Wagon        = ['a'..'i'];
     Spoor        = ['x'..'z'];
     Configuratie = ARRAY Wagon OF Spoor;
VAR  wn           : Configuratie;
```

Een configuratie is een afbeelding van elk van de wagons op de sporen. De index van een tabelelement stemt overeen met een wagon. Voor het gegeven voorbeeld geldt de configuratie :

yxxyzzyzy

De functieprocedure NietOpSpoor levert als waarde het derde spoor, verschillend van de twee argumenten.

```
PROCEDURE NietOpSpoor(    s1 : Spoor
                          s2 : Spoor) : Spoor;
BEGIN
RETURN
  CHR(ORD('x') + ORD('y') + ORD('z') - ORD(s1) - ORD(s2))
END NietOpSpoor;
```

De procedure Verplaats verplaatst een wagon naar een gegeven spoor. De essentie van deze procedure is de aanpassing van de configuratie.

```
PROCEDURE Verplaats(    w          : Wagon;
                        bestemming : Spoor);
BEGIN
wn[w] := bestemming;
(* geef de nieuwe toestand weer *)
END Verplaats;
```

In deze procedure definiëren we ook een aantal opdrachten voor de weergave van de actuele toestand. We geven hiervoor twee oplossingen : de eerste oplossing schrijft een bericht voor elke verplaatsing, zoals in de oplossing van het probleem 'Torens van Hanoi'. De tweede oplossing tekent steeds de volledige configuratie op het scherm zodat we het formeren van de trein als een tekenfilm kunnen bekijken.

De procedure VerwerkTrein zorgt voor het formeren van de trein. Het algoritme voor deze procedure is hetzelfde als dat voor de Torens van Hanoi.

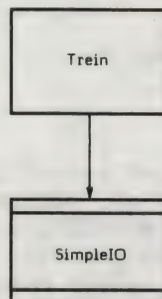
```
PROCEDURE VerwerkTrein(    w          : Wagon;
                           bestemming : Spoor);

  PROCEDURE Pred(w : Wagon) : Wagon;
  BEGIN
  RETURN CHR(ORD(w) - 1)
  END Pred;

BEGIN
IF w = 'a'
THEN
  IF wn['a'] # bestemming
  THEN
    Verplaats('a', bestemming)
  END
ELSE
  IF wn[w] <> bestemming
  THEN
    VerwerkTrein(Pred(w), NietOpSpoor(wn[w], bestemming));
    Verplaats(w, bestemming);
  END;
  VerwerkTrein(Pred(w), bestemming)
END
END VerwerkTrein;
```

Oplossing 1 : weergave van de verplaatsingen

In deze oplossing gebruiken we de bewerkingen WriteString, WriteChar, ReadChar, en WriteLn van de module SimpleIO. Het abstractieschema is dan :



figuur 7.2 Trein : oplossing 1

```
MODULE Trein;
FROM SimpleIO IMPORT WriteString, WriteChar, WriteLn,
                     ReadChar;

TYPE Wagon = ['a'..'i'];
     Spoor = ['x'..'z'];
     Configuratie = ARRAY Wagon OF Spoor;
VAR  wn : Configuratie;


PROCEDURE NietOpSpoor(    s1 : Spoor;
                          s2 : Spoor) : Spoor;
BEGIN
RETURN
  CHR(ORD('x') + ORD('y') + ORD('z') - ORD(s1) - ORD(s2))
END NietOpSpoor;


PROCEDURE Verplaats(    w           : Wagon;
                        bestemming  : Spoor);
BEGIN
WriteChar(w);
WriteString(' : ');
WriteChar(wn[w]);
WriteString('-->');
WriteChar(bestemming);
WriteLn;
wn[w] := bestemming
END Verplaats;


PROCEDURE VerwerkTrein(    w           : Wagon;
                           bestemming  : Spoor);

  PROCEDURE Pred(w : Wagon) : Wagon;
  BEGIN
  RETURN CHR(ORD(w) - 1)
  END Pred;

BEGIN
IF w = 'a'
THEN
  IF wn['a'] # bestemming
  THEN
    Verplaats('a', bestemming)
  END
ELSE
```

```
  IF wn[w] <> bestemming
  THEN
    VerwerkTrein(Pred(w), NietOpSpoor(wn[w], bestemming));
    Verplaats(w, bestemming);
  END;
  VerwerkTrein(Pred(w), bestemming)
END
END VerwerkTrein;

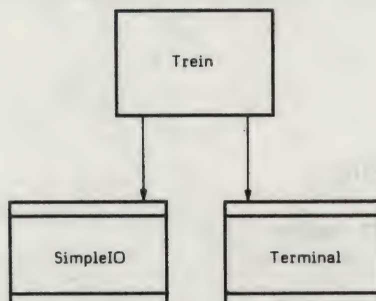

VAR bestemming : Spoor;
    t          : Wagon;
BEGIN
(* lees de startconfiguratie *)
FOR t := 'a' TO 'i' DO
  ReadChar(wn[t])
END;

(* lees bestemming *)
ReadChar(bestemming);

(* vorm trein *)
VerwerkTrein('i', bestemming)
END Trein.
```

Oplossing 2 : weergave van de actuele configuratie

In deze oplossing gebruiken we ook nog bewerkingen voor de besturing van het beeldscherm. Deze bewerkingen worden gedefinieerd in de module Terminal. We beperken ons hier tot de bewerking EraseScreen : EraseScreen wist het scherm en plaatst de cursor in de linkerbovenhoek. Elke nieuwe configuratie wordt dan steeds vanaf deze positie geschreven. De uitvoer kan nog beter worden verzorgd met de overige bewerkingen van Terminal. Het abstractieschema voor deze oplossing is nu :



figuur 7.3 Trein : oplossing 2

De bewerking EraseScreen van de module Terminal is beschikbaar in de programmamodule na de declaratie :

```
FROM Terminal IMPORT EraseScreen;
```

De procedure Verplaats wordt vervangen door de procedure :

```
PROCEDURE Verplaats(    w          : Wagon;
                        bestemming : Spoor);
VAR i     : Wagon;
    spoor : Spoor;

BEGIN
wn[w] := bestemming;
EraseScreen;
FOR spoor := 'x' TO 'z' DO
  WriteChar(spoor);
  WriteString(' : ');
  FOR i := 'a' TO 'i' DO
    IF wn[i] = spoor
    THEN
      WriteChar(i)
    END
  END;
  WriteLn
END
END Verplaats;
```

7.2 Indirecte recursie

Alle voorbeelden die we nu tot hebben beschreven zijn direct recursief : de procedure roept zichzelf expliciet aan. Een andere vorm is wederzijdse of indirecte recursie : een procedure A roept een procedure B aan die op haar beurt weer rechtstreeks of via een omweg procedure A aanroept. Schematisch is de vorm :

```
PROCEDURE A (...);
BEGIN

  B(...)

END A;
```

```
PROCEDURE B (...);
BEGIN

  A(...)

END B;
```

Bewerkingen op datastructuren die zelf recursief zijn gedefinieerd, worden dikwijls geformuleerd met indirecte recursieve procedures. Voorbeelden hiervan worden later beschreven.

Literatuur

Alagic S., Arbib M.A., 'The Design of Well-Structured Programs', Springer-Verlag, 1978

Ford G.A., Wiener R., 'Modula-2, A Software Development Approach, Recursion',
Wiley, 1985

Meelhuysen W., 'Wegwijs in PL/1',
Samson, 1979

Stone R. G., 'Points Recurring, The History of the Railway Problem',
Sigplan

Wirth N., 'Algorithms + Data Structures = Programs',
Prentice Hall, 1976

Wirth, N., 'Programming in Modula-2',
Springer-Verlag, derde verbeterde druk, 1985

Oefeningen

1. De Fibonacci-getallen worden gedefinieerd als :

$$f_1 = 0$$

$$f_2 = 1$$

$$f_n = f_{n-1} + f_{n-2}$$

Onderzoek de volgende recursieve functieprocedures voor het berekenen van het n-de Fibonacci-getal. Wat is fout en waarom ?

a.
```
PROCEDURE NFib(n : INTEGER) : INTEGER;
BEGIN
RETURN NFib(n - 1) + NFib(n - 2)
END NFib;
```

b.
```
PROCEDURE NFib(n : INTEGER) : INTEGER;
BEGIN
IF n = 1
THEN
  RETURN Ø
ELSIF (n = 2) OR (n = 3)
THEN
  RETURN 1
ELSE
  RETURN NFib(n - 1) + NFib(n - 2)
END
END NFib;
```

2. Formuleer een correcte oplossing voor de functieprocedure NFib.

3. Schrijf een functieprocedure Som voor het berekenen van de som van n elementen van tabel a. Het elementtype is CARDINAL. De procedurekop is :

```
PROCEDURE Som(    a : ARRAY OF CARDINAL;
                  n : CARDINAL) : CARDINAL;
```

4. Schrijf een functieprocedure voor het berekenen van het aantal mogelijke partities van een getal. We definiëren een partitie van een positief getal n als een uitdrukking a + b + c + ... + k waarvan de waarde gelijk is aan n. Bijvoorbeeld de partities van 5 zijn :

5
4 + 1
3 + 2
3 + 1 + 1
2 + 2 + 1
2 + 1 + 1 + 1
1 + 1 + 1 + 1 + 1

5. Schrijf een functieprocedure PlaatsLineair voor het opzoeken van de plaats van een element in een tabel a met n elementen. Het elementtype is CARDINAL.

6. Schrijf een functieprocedure Produkt voor het berekenen van het produkt van twee getallen m en n met type CARDINAL.

7. Schrijf een functieprocedure Macht voor het berekenen van de machtsverheffing m^n. Het type van m en n is CARDINAL.

8. Schrijf een functieprocedure voor het berekenen van de vierkantswortel van een getal. Gebruik de volgende formule :
Vierkantswortel(n, benadering, epsilon) =

- benadering, indien
  | benadering^2 - n | < epsilon
- Vierkantswortel(n, (a * a + n) / (2 * a), epsilon)

9. Schrijf een functieprocedure MinimumTabel voor het bepalen van het kleinste element van een tabel.

10. Beschouw het spoorwegprobleem. Veronderstel een gemeenschappelijk spoor met voldoende plaats voor de volledige trein. Formuleer een nieuwe oplossing waarbij rekening wordt gehouden met de wagons die reeds na elkaar op hetzelfde spoor staan.

11. Voor het spoorwegprobleem wordt in deze opgave niet de bestemming opgegeven, maar wel een geldige eindconfiguratie. De invoer voor het programma is bijgevolg een begin- en een eindconfiguratie. Wijzig het programma.

8 Wijzers en dynamische variabelen

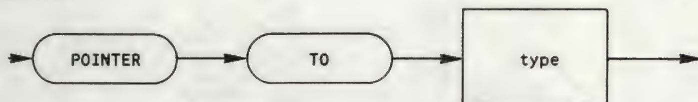
Bij veel toepassingen is de omvang of de structuur van een verzameling gegevens niet vooraf bekend omdat deze kunnen variëren bij elke uitvoering van het programma. Tabellen en records zijn statisch : zij hebben een vaste structuur en zijn voor deze toepassingen ongeschikt. Voor een variabele met een statische structuur berekent de compiler de geheugenruimte en bij de uitvoering wordt een geheugengebied aan deze variabele toegekend. De levensduur van de variabele is gelijk aan de levensduur van het blok, waarin de variabele is gedeclareerd. Ook het adres van de geheugenzone wordt vooraf berekend. Een element van een tabel of een veld van een record is toegankelijk na de berekening van de verplaatsing ten opzichte van het adres van het geheugengebied. De verplaatsing is een functie van de index van het element of van de naam van het veld en van de geheugenruimte die aan de voorafgaande componenten van de structuur is toegekend. Deze eigenschappen verhinderen dat variabelen met een statische structuur tijdens de uitvoering kunnen veranderen in grootte of organisatie.

8.1 Het wijzertype en de wijzervariabele

Modula-2 bevat een wijzer- of POINTER-mechanisme waarbij de geheugenruimte tijdens de uitvoering van het programma aan variabelen kan worden toegekend. Door middel van een wijzertype construeren we nieuwe typen, dit zoals met ARRAY, SET of RECORD. Elke wijzer is gebonden aan een basistype. De waardenverzameling van een wijzertype is de verzameling geheugenadressen voor de objecten met het gegeven basistype. De syntaxis voor een wijzerdeclaratie is :

WijzerType = 'POINTER' 'TO' type

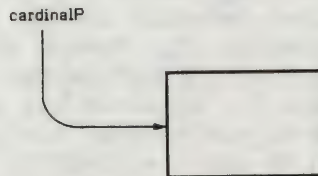
WijzerType



Het basistype is :
- een voorgedefinieerd standaardtype zoals CARDINAL, CHAR of INTEGER;
- een nieuw type dat we vooraf hebben gedefinieerd, bijvoorbeeld een ARRAY, SET of een deelinterval;
- een type dat na het wijzertype in hetzelfde blok wordt gedefinieerd. Dit is de enige uitzondering op de algemene regel dat naar een object slechts mag worden verwezen na de declaratie.

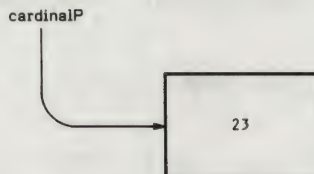
De voorgedefinieerde constante NIL behoort steeds tot de waardenverzameling van elk wijzertype. Deze waarde geeft aan dat de huidige waarde van de wijzervariabele niet overeenstemt met een adres van een variabele. We illustreren de betekenis van het wijzertype in de volgende figuur. Elke pijl betekent 'verwijst naar' of 'is het adres van'.

```
TYPE CardinalPointer = POINTER TO CARDINAL;
VAR  cardinalP       : CardinalPointer;
```



figuur 8.1 De betekenis van het wijzertype

Voor het wijzertype zijn gedefinieerd : de toekenningsopdracht, de operatoren gelijk '=' en ongelijk '#' voor het vergelijken van wijzervariabelen en ten slotte de dereferentieoperator. De dereferentieoperator is een unaire postfix-operator : de operator heeft slechts één operand en wordt na de operand geplaatst. Het symbool voor de operator is het opwaartse pijltje '↑' of het '^'-teken ('caret'-teken). De waarde van een wijzervariabele gevolgd door de dereferentieoperator is de waarde van het object waarnaar de wijzer verwijst. Het object is een naamloze variabele en is slechts via de wijzer toegankelijk.



figuur 8.2 De dereferentie-operator

De waarde van cardinalP↑ is 23.

8.2 Het creëren en vrijgeven van naamloze variabelen

De naamloze variabelen worden niet gedeclareerd in de programmatekst maar worden gecreëerd en verwijderd met de standaardprocedures NEW en DISPOSE. De procedures NEW en DISPOSE zijn generieke procedures : het aantal actuele parameters en het type van de parameters zijn afhankelijk van de betekenis in de tekst waar de procedures worden gebruikt. In zijn eenvoudigste vorm heeft de NEW-procedure een variabele parameter met een wijzertype. Het effect van de procedure is :

- een geheugenruimte wordt toegewezen aan een naamloze variabele met het basistype van de actuele wijzerparameter;
- de waarde van de actuele parameter wordt het adres van de geheugenruimte.

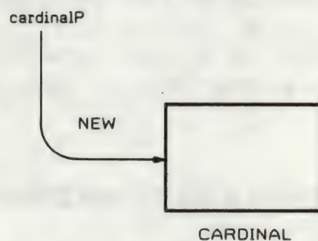
Voorbeeld :

```
TYPE CardinalPointer = POINTER TO CARDINAL;
VAR  cardinalP       : CardinalPointer;
```

De opdracht

```
NEW(cardinalP)
```

creëert de geheugenruimte voor een naamloze variabele met type CARDINAL.



figuur 8.3 Het creëren van een naamloze variabele

Na de geheugentoewijzing verwijzen we naar de variabele met de dereferentieoperator :

```
cardinalP↑ := 23
```

Deze opdracht kent aan de naamloze variabele de waarde 23 toe.

De levensduur van een naamloze variabele is niet gebonden aan de levensduur van het blok waarin de variabele is gecreëerd noch aan de levensduur van het blok waarin de wijzer is gedeclareerd.

Voorbeeld :

```
MODULE A;
...
  PROCEDURE P1;
  VAR p : POINTER TO CARDINAL;

  BEGIN
  NEW(p);
  p↑ := 23;
  END P1;
...
BEGIN
P1;
...
END A.
```

Het bereik van de wijzervariabele p is beperkt tot de procedure P1. De levensduur van p is dus beperkt tot de tijd dat procedure P1 actief is. Procedure P1 creëert een naamloze variabele met type CARDINAL en kent hieraan de waarde 23 toe. Aan het einde van de activiteit van P1 eindigt ook de levensduur van de wijzervariabele p. De naamloze variabele blijft wel bestaan maar is niet meer toegankelijk.

Met de procedure DISPOSE geven we de geheugenruimte die aan een naamloze variabele is toegewezen, weer vrij. In de eenvoudigste vorm bevat de aanroep een actuele parameter met een wijzertype. Het effect van de aanroep is :
- de geheugenruimte waarnaar de wijzer verwijst wordt vrijgegeven;
- de nieuwe waarde van de wijzer is niet gedefinieerd.

8.3 De implementatie van NEW en DISPOSE

De standaardprocedures NEW en DISPOSE worden gedefinieerd met behulp van de bewerkingen ALLOCATE en DEALLOCATE van het abstracte datatype Storage. De implementatie van deze bewerkingen is afhankelijk van de omgeving, de computer of het besturingssysteem waarop de Modula-2 programma's draaien. De module Storage behoort tot de standaardbibliotheek en levert de bewerkingen voor het dynamisch beheer van het geheugen. De definitie van de procedures ALLOCATE en DEALLOCATE is :

```
PROCEDURE ALLOCATE  (VAR adres   : ADDRESS;
                         grootte : CARDINAL);
PROCEDURE DEALLOCATE(VAR adres   : ADDRESS;
                         grootte : CARDINAL);
```

Bij de aanroep is de waarde van de eerste parameter de wijzer van het basistype. Het type ADDRESS is gedefinieerd in de systeem-afhankelijke module SYSTEM als

```
ADDRESS = POINTER TO WORD
```

Een woord is een elementaire geheugencel. Een waarde van het type ADDRESS kan aan elke variabele van een willekeurig wijzertype worden toegekend. De hoeveelheid geheugen bepalen we met de procedure TSIZE. Deze procedure wordt ook gedefinieerd in de module SYSTEM :

```
PROCEDURE TSIZE(TypeIdentifier) : CARDINAL;
```

Voorbeeld :

```
TYPE CardinalPointer = POINTER TO CARDINAL;
VAR  cardP           : CardinalPointer;
```

De opdracht

```
NEW(cardP)
```

wordt door de compiler omgezet in de opdracht

```
ALLOCATE(cardP, TSIZE(CARDINAL))
```

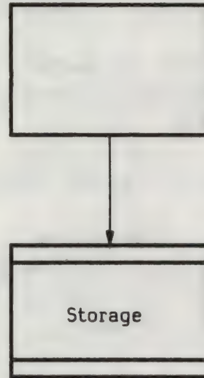
In een programma kunnen we de beide opdrachten gebruiken voor het creëren van een naamloze variabele. Een analoge redenering geldt voor de procedures DISPOSE en DEALLOCATE. De implementatie van NEW en DISPOSE met ALLOCATE en DEALLOCATE heeft tot gevolg dat we deze laatste procedures in een module beschikbaar moeten stellen met de declaratie :

```
FROM Storage IMPORT ALLOCATE, DEALLOCATE;
```

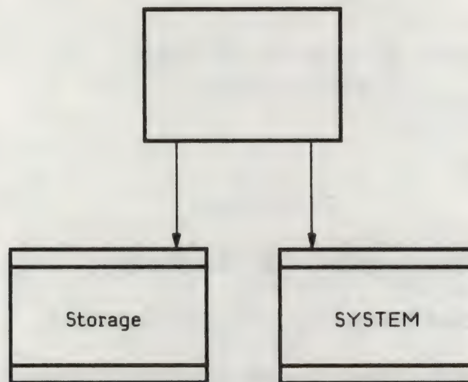
Als we het geheugen beheren met de procedures ALLOCATE en DEALLOCATE in plaats van NEW en DISPOSE, gebruiken we de procedure TSIZE, tenzij de grootte van de geheugenruimte bekend is. Hiervoor declareren we dan :

```
FROM SYSTEM IMPORT TSIZE;
```

We stellen beide oplossingsmethoden voor met de volgende abstractieschema's.



figuur 8.4 Dynamisch beheer met NEW en DISPOSE



figuur 8.5 Dynamisch beheer met ALLOCATE en DEALLOCATE

8.4 Het toewijzen en vrijgeven van records met varianten

Een recordstructuur kan met varianten worden gedeclareerd. De hoeveelheid geheugenruimte hoeft voor elk van deze varianten niet dezelfde te zijn. We kunnen deze hoeveelheid laten berekenen door het toevoegen van een of meer parameters aan de parameterlijst bij de aanroep van NEW of DISPOSE.

Voorbeeld :

```
TYPE Indeling  = (leeg, klein, groot);
     Kader     = RECORD
       lengte  : CARDINAL;
       breedte : CARDINAL;
       CASE indeling : Indeling OF
         leeg  :
       | klein : teken : CHAR;
       | groot : ARRAY [1..10],[1..10] OF CHAR
       END
     END;
     KaderPointer = POINTER TO Kader;

VAR  kaderP       : KaderPointer;
```

Voor het recordtype Kader gelden drie verschillende grootten, afhankelijk van de keuze 'indeling'. Met de opdracht

NEW(kaderP)

wordt een geheugenruimte met de maximale grootte toegewezen. Als we echter weten dat met minder ruimte volstaan kan worden, specificeren we de etiketwaarde voor de variant bij de aanroep van de procedure NEW. De kleinste geheugenruimte wordt toegekend met de opdracht

NEW(kaderP, leeg)

Daarna registeren we de keuze in het etiketveld :

kaderP↑.indeling := leeg;

De geheugenruimte wordt weer vrijgegeven met de procedure DISPOSE. Bij deze aanroep moet de ruimte worden vrijgegeven die daarvóór is toegekend. We vermelden de waarde van het etiketveld voor de variant :

DISPOSE(kaderP, kaderP↑.indeling)

Verscheidene varianten worden aangegeven door een derde of meer parameters bij de aanroep van de procedure NEW of DISPOSE.

8.5 Toepassingen

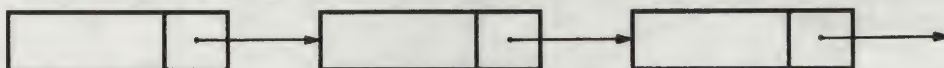
We illusteren het gebruik van het wijzertype met twee voorbeelden : de constructie van een lijst en van een dynamische matrix. Bij een lijst wijzigt de gegevensstructuur tijdens het uitvoeren van het programma. Bij de dynamische matrix kunnen we de omvang van de structuur tijdens het uitvoeren vastleggen.

Lijsten

Een lijst bestaat uit een verzameling elementen. Een knooppunt is een element met een gegeven type. We definiëren de lijst als een recursieve gegevensstructuur :

1. een lege verzameling knooppunten is een lijst (lege lijst);
2. als k een knooppunt is en s een lijst, dan vormt het geordende paar (k, s) opnieuw een lijst. k is de kop (Engels : header) van de lijst, s de staart (Engels : tail).

De grafische voorstelling van een lijst is :



figuur 8.6 Grafische voorstelling van een lijst

Elke pijl in deze voorstelling kunnen we in een gegevensstructuur voorstellen door een wijzer. Elk knooppunt van de lijst is dus samengesteld uit twee delen : een informatiedeel en een verbindingsdeel (een wijzer). Als de waarde van het verbindingsdeel gelijk is aan NIL, dan is het knooppunt het laatste van de lijst.

We declareren een type voor een knooppunt. Hierbij veronderstellen we dat het type Informatie vooraf is gedefinieerd.

```
TYPE Verbinding = POINTER TO Knooppunt;
     Knooppunt  = RECORD
       informatie : Informatie;
       verbinding : Verbinding
     END;
```

De declaratie van Knooppunt is recursief : het veld 'verbinding' verwijst indirect naar de declaratie van Knooppunt.

De kop van een lijst verwijst naar het eerste knooppunt. We declareren de kop en een knooppunt als volgt :

```
VAR kop       : Verbinding;
    knooppunt : Verbinding;
```

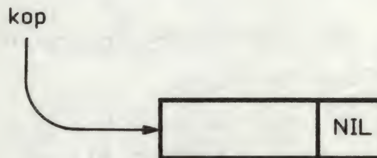
Een lijst wordt dynamisch opgebouwd. We beginnen met een lege lijst. Voor de lege lijst voeren we de toekenningsopdracht uit :

```
kop := NIL
```

Een knooppunt wordt aan de lege lijst toegevoegd met de volgende opdrachten :

```
NEW(knooppunt);
kop := knooppunt;
WITH knooppunt↑ DO
  (* verwerking van de informatie *)
  verbinding := NIL
END;
```

We beschikken nu over een lijst met één knooppunt. De variabele kop verwijst naar het begin van de lijst, het verbindingsveld van het (laatste) knooppunt is gelijk aan NIL.



figuur 8.7 Een lijst met één knooppunt

We definiëren nu enkele eenvoudige procedures voor bewerkingen met lijsten : het creëren van een knooppunt, het toevoegen van een knooppunt aan het einde van de lijst en het verwijderen van een gegeven knooppunt. We definiëren de procedures voor een knooppunt waarvan het informatiedeel bestaat uit een getal met type CARDINAL.

```
TYPE Verbinding = POINTER TO Knooppunt;
     Knooppunt  = RECORD
       informatie : CARDINAL;
       verbinding : Verbinding
     END;

PROCEDURE NieuweKnoop(    informatie : CARDINAL;
                      VAR knoop      : Verbinding);

BEGIN
NEW(knoop);
knoop↑.informatie := informatie;
END NieuweKnoop;

PROCEDURE VoegKnoopToe(    knoop : Verbinding;
                       VAR kop   : Verbinding);
VAR p : Verbinding;
BEGIN
(* voeg het knooppunt toe aan de lijst *)
IF kop = NIL
THEN
  (* de lijst is nog leeg *)
  kop := knoop
ELSE
  (* zoek het einde van de lijst *)
  p := kop;
  WHILE p↑.verbinding # NIL DO
    p := p↑.verbinding
  END;

  (* pas de wijzer aan van het laatste knooppunt *)
  p↑.verbinding := knoop
END;

(* de knoop wordt het laatste knooppunt *)
knoop↑.verbinding := NIL
END VoegKnoopToe;

PROCEDURE WisKnoop(    informatie : CARDINAL;
                   VAR kop        : Verbinding);
VAR p, q : Verbinding;
BEGIN
(* zoek het eerste knooppunt met de gegeven informatie *)
p := kop;
q := NIL;
WHILE (p # NIL) AND (p↑.informatie # informatie) DO
  q := p;
  p := p↑.verbinding
END;
```

```
(* stop de verwerking als het knooppunt niet is gevonden *)
IF p = NIL
THEN
  RETURN
END;

(* wis het knooppunt *)
IF p = kop
THEN
  (* wis het eerste knooppunt *)
  kop := p↑.verbinding
ELSE
  q↑.verbinding := p↑.verbinding
END;

(* geef geheugenruimte vrij *)
DISPOSE(p)
END WisKnoop;
```

Op lijsten kunnen nog talrijke andere bewerkingen worden gedefinieerd. We geven hiervan enkele voorbeelden :

- een knooppunt toevoegen aan een lijst (aan het begin, aan het einde of in het midden van de lijst);
- een knooppunt opzoeken in de lijst;
- een knooppunt verwijderen uit de lijst (aan het begin, aan het einde, juist voor of na een gegeven knooppunt);
- twee lijsten samenvoegen;
- een lijst kopiëren.

Dynamische tabellen

We beschouwen de declaraties

```
TYPE T1 = ARRAY [1.100] OF CARDINAL;
     T2 = ARRAY ['a'..'z'] OF CARDINAL;
     T3 = ARRAY (a, b, c, d) OF CARDINAL;
```

Met behulp van een open-array-parameter en het gebruik van de standaardfunctie HIGH kunnen we algemene procedures formuleren voor het behandelen van tabellen met het type T1, T2 of T3, bijvoorbeeld :

```
PROCEDURE Som(a : ARRAY OF CARDINAL) : CARDINAL;
VAR i   : CARDINAL;
    s   : CARDINAL;
```

```
BEGIN
s := 0;
FOR i := 0 TO HIGH(a) DO
  INC(s, a[i])
END;
RETURN s
END Som;

VAR t1 : T1;
    t2 : T2;
    t3 : T3;
```

De aanroepen Som(t1), Som(t2) en Som(t3) berekenen de som van de elementen respectievelijk voor de tabellen met type T1, T2 en T3.

Een open-array-parameter is echter niet gedefinieerd voor tabellen met verschillende dimensies. We definiëren nu een model met een dynamisch tabeltype opdat we ook voor tweedimensionale tabellen (matrices) algemene procedures kunnen formuleren. In het model houden we rekening met de volgende karakteristieken van een matrix :

1. de onder- en bovengrens van de rij-index;
2. de onder- en bovengrens van de kolomindex;
3. het basistype van een matrixelement. In dit voorbeeld beschouwen we uitsluitend het basistype CARDINAL.

Uit deze karakteristieken leiden we het aantal rijen, kolommen en elementen af :

aantal rijen = bovengrens rij-index - ondergrens rijindex + 1
aantal kolommen = bovengrens kolomindex - ondergrens kolomindex + 1
aantal elementen = aantal rijen * aantal kolommen

De geheugenruimte voor de opslag van de matrix is afhankelijk van het aantal elementen en van het basistype van een element. Deze geheugenruimte wordt dynamisch toegekend. De geheugenruimte voor de opslag van de matrixkarakteristieken is constant en kan statisch worden gedeclareerd.

We definiëren het type Matrix als :

```
CONST aantalDimensies = 2;
TYPE  DimensieIndex   = [1..aantalDimensies];
      Grens           = (ondergrens, bovengrens);
      Dimensie        = ARRAY DimensieIndex, Grens OF INTEGER;

      Matrix          = RECORD
        dimensie : Dimensie;
        element  : POINTER TO CARDINAL
      END;
```

Deze definitie is algemeen en kan later eenvoudig worden aangepast voor een aantal dimensies groter dan twee. Ook definiëren we de volgende bewerkingen voor het beheer van dit type matrix :

- het creëren van een matrix met de procedure CreeerMatrix :

```
PROCEDURE CreeerMatrix(    dim : Dimensie;
                       VAR m   : Matrix);
```

- het vrijgeven van de dynamische geheugenruimte van een matrix :

```
PROCEDURE GeefElementVrij(VAR m : Matrix);
```

- het bepalen van de bovengrens van een dimensie :

```
PROCEDURE Hoog(    m : Matrix;
                   k : DimensieIndex) : INTEGER;
```

- het bepalen van de ondergrens van een dimensie :

```
PROCEDURE Laag(    m : Matrix;
                   k : DimensieIndex) : INTEGER;
```

De geheugenruimte waarin de matrixelementen worden bewaard is één-dimensionaal. We beelden de matrix af op deze structuur volgens de rij-per-rij-organisatie : eerst worden de elementen van de eerste rij opgeslagen, daarna de elementen van de tweede rij enzovoort. De verplaatsing van een element met indices i en j ten opzichte van het begin van de toegekende geheugenruimte wordt berekend volgens de uitdrukking :

```
verplaatsing element[i, j] =
   ((i - ondergrens rij-index) * aantal kolommen +
    (j - ondergrens kolomindex)) * grootte van een element
```

We definiëren de procedure verplaatsing als

```
PROCEDURE Verplaatsing(    m     : Matrix;
                           rij   : INTEGER;
                           kolom : INTEGER) : CARDINAL;
```

Via het adres van de geheugenruimte en de verplaatsing wordt het adres van het element berekend :

```
adres element := adres geheugenruimte + verplaatsing
```

Deze uitdrukking is echter onmogelijk in Modula-2 : de uitdrukking bevat operanden met een verschillend type en bovendien kunnen met

wijzervariabelen geen rekenkundige bewerkingen worden uitgevoerd. Het adres van een variabele is systeemafhankelijk. Met de module SYSTEM kunnen we ook met systeemafhankelijke objecten werken. In de module SYSTEM is het type ADDRESS gedefinieerd. We veronderstellen voor dit type de volgende eigenschappen :

- ADDRESS = POINTER TO WORD; een variabele met dit type is een adres van een willekeurige geheugencel;
- een waarde van het type ADDRESS past bij een willekeurig wijzertype bij het uitvoeren van een toekenningsopdracht;
- voor het type ADDRESS zijn de rekenkundige operatoren '+' en '-' en de vergelijkingsoperatoren '=' en '#' gedefinieerd;
- de standaardprocedures INC en DEC gelden ook voor het type ADDRESS;
- een gemengde uitdrukking met operanden met type ADDRESS en CARDINAL is toegelaten.

De uitdrukking voor de berekening van het adres van een element met indices i en j wordt nu :

```
adres element := ADDRESS(adres geheugenruimte) + verplaatsing
```

Het resultaattype van deze uitdrukking is ook ADDRESS. Dit type past bij het wijzertype van het linkerlid bij het uitvoeren van een toekenningsopdracht.

Tot slot kunnen we nu de bewerkingen definiëren voor de manipulatie van een afzonderlijk matrixelement : SchrijfElement en WaardeElement.

```
PROCEDURE SchrijfElement(    m      : Matrix;
                             i      : INTEGER;
                             j      : INTEGER;
                             waarde : CARDINAL);
```

De procedure SchrijfElement registreert de waarde van een matrixelement in de juiste geheugenlocatie.

```
PROCEDURE WaardeElement (    m      : Matrix;
                             i      : INTEGER;
                             j      : INTEGER) : CARDINAL;
```

De functieprocedure WaardeElement levert de waarde van het matrixelement met de indices i en j.

We geven nu de implementatie voor deze bewerkingen :

```
FROM SYSTEM  IMPORT TSIZE, ADDRESS;
FROM Storage IMPORT ALLOCATE, DEALLOCATE;

PROCEDURE CreeerMatrix(    dim : Dimensie;
                       VAR m   : Matrix);
BEGIN
WITH m DO
  dimensie := dim;
  ALLOCATE(element,
    TSIZE(CARDINAL) * CARDINAL((Hoog(m, 1) - Laag(m, 1) + 1) *
                               (Hoog(m, 2) - Laag(m, 2) + 1)))
END
END CreeerMatrix;

PROCEDURE GeefElementVrij(VAR m : Matrix);
BEGIN
WITH m DO
  DEALLOCATE(element,
    TSIZE(CARDINAL) * CARDINAL((Hoog(m, 1) - Laag(m, 1) + 1) *
                               (Hoog(m, 2) - Laag(m, 2) + 1)))
END
END GeefElementVrij;

PROCEDURE Hoog(    m : Matrix;
                   k : DimensieIndex) : INTEGER;
BEGIN
RETURN m.dimensie[k, bovengrens]
END Hoog;

PROCEDURE Laag(    m : Matrix;
                   k : DimensieIndex) : INTEGER;
BEGIN
RETURN m.dimensie[k, ondergrens]
END Laag;


PROCEDURE Verplaatsing(    m     : Matrix;
                           rij   : INTEGER;
                           kolom : INTEGER) : CARDINAL;
BEGIN
RETURN CARDINAL(
  (rij - Laag(m, 1)) * (Hoog(m, 2) - Laag(m, 2) + 1) +
  kolom - Laag(m, 2) + 1)
  * TSIZE(CARDINAL)
END Verplaatsing;
```

```
PROCEDURE SchrijfElement(    m      : Matrix;
                             i      : INTEGER;
                             j      : INTEGER;
                             waarde : CARDINAL);
VAR adres : POINTER TO CARDINAL;

BEGIN
adres := ADDRESS(m.element) + Verplaatsing(m, i, j);
adres↑ := waarde
END SchrijfElement;


PROCEDURE WaardeElement(    m      : Matrix;
                            i      : INTEGER;
                            j      : INTEGER) : CARDINAL;
VAR adres : POINTER TO CARDINAL;

BEGIN
adres := ADDRESS(m.element) + Verplaatsing(m, i, j);
RETURN adres↑
END WaardeElement;
```

Met de gegevensstructuur Matrix en de bewerkingen die hierop zijn gedefineerd beschikken we nu over een datatype 'matrix met basistype CARDINAL'. Met dit type definiëren we enkele algemene procedures LeesTabel, SchrijfTabel en SomTabel. De in- en uitvoerbewerking voor de in- en uitvoer van de elementen leveren we via een procedureparameter. De procedures LeesTabel en SchrijfTabel zijn dan onafhankelijk van de oorsprong van de elementen en van de wijze waarop de in- en uitvoer plaatsvindt. De procedures LeesTabel en SchrijfTabel bevatten ook de parameter succes met type BOOLEAN. De waarde van succes is TRUE als de opdracht met de tabel met succes is uitgevoerd, anders is de waarde FALSE.

```
TYPE InProcedure  = PROCEDURE(VAR CARDINAL, VAR BOOLEAN);
     UitProcedure = PROCEDURE(    CARDINAL, VAR BOOLEAN);

PROCEDURE LeesTabel(    m      : Matrix;
                    VAR succes : BOOLEAN;
                        in     : InProcedure);
(* inlezen van de elementen van een willekeurige matrix
   met basistype CARDINAL *)

VAR i, j    : INTEGER;
    element : CARDINAL;
```

```
BEGIN
FOR i := Laag(m, 1) TO Hoog(m, 1) DO
  FOR j := Laag(m, 2) TO Hoog(m, 2) DO
    in(element, succes);
    IF NOT succes
    THEN
      RETURN
    END;
    SchrijfElement(m, i, j, element)
  END
END
END LeesTabel;


PROCEDURE DrukTabel(    m      : Matrix;
                    VAR succes : BOOLEAN;
                        uit    : UitProcedure);
(* uitvoeren van de elementen van een willekeurige matrix
   met basistype CARDINAL *)
VAR i, j    : INTEGER;
    element : CARDINAL;

BEGIN
FOR i := Laag(m, 1) TO Hoog(m, 1) DO
  FOR j := Laag(m, 2) TO Hoog(m, 2) DO
    uit(WaardeElement(m, i, j), succes);
    IF NOT succes
    THEN
      RETURN
    END
  END
END
END DrukTabel;


PROCEDURE SomTabel (    m         : Matrix;
                        n         : Matrix;
             (* uit *) resultaat : Matrix);
(* berekenen van de som van de matrices m + n *)
VAR i, j    : INTEGER;

BEGIN
FOR i := Laag(m, 1) TO Hoog(m, 1) DO
  FOR j := Laag(m, 2) TO Hoog(m, 2) DO
    SchrijfElement(resultaat, i, j,
             WaardeElement(m,i,j) + WaardeElement(n,i,j))
  END
END
END SomTabel;
```

Met het volgende programmafragment creëren we de geheugenruimte voor de elementen van de matrices a, b en c. Daarna worden a en b gelezen, wordt c berekend en tot slot wordt het resultaat afgedrukt. We veronderstellen dat we de procedures ReadElement en WriteElement, met definities

```
PROCEDURE ReadElement (VAR element : CARDINAL;
                       VAR succes  : BOOLEAN);
PROCEDURE WriteElement(    element : CARDINAL;
                       VAT succes  : BOOLEAN);
```

kunnen gebruiken.

```
VAR a, b, c     : Matrix;
    dim         : Dimensie;
    ok          : BOOLEAN;

(* invullen van de dimensies *)
dim[1, ondergrens] := -3;
dim[1, bovengrens] := 10;
dim[2, ondergrens] := -10;
dim[2, bovengrens] := 10;

(* creëren van de dynamische geheugenruimte *)
CreeerMatrix(dim, a);
CreeerMatrix(dim, b);
CreeerMatrix(dim, c);

(* inlezen van de tabel a met de procedure ReadCard *)
LeesTabel(a, ok, ReadCard);
IF NOT ok
THEN
  HALT
END;

(* inlezen van de tabel b met de procedure ReadElement *)
LeesTabel(b, ok, ReadElement);
IF NOT ok
THEN
  HALT
END;

(* verwerken van de matrices *)
SomTabel(a, b, c);

(* uitvoeren van het resultaat *)
DrukTabel(c, ok, WriteElement)
```

We hebben in dit voorbeeld de gegevensstructuur en de bewerkingen gedefinieerd voor een matrix waarvan het aantal elementen dynamisch kan worden gewijzigd. Als we deze gegevensstructuur en de bewerkingen onzichtbaar kunnen maken voor de omgeving, beschikken we over het abstracte datatype 'matrix met basistype CARDINAL'. Verdere abstracties zijn mogelijk, bijvoorbeeld een willekeurig basistype of een groter aantal dimensies. De definitie en de implementatie van abstracte datatypen wordt beschreven in de volgende hoofdstukken. De generalisering van het basistype wordt beschreven in deel twee bij de definitie van algemene abstracte datatypen.

Literatuur

van Berne, Duijvestijn en van der Hoeven, 'Functionele talen', Informatie, oktober 1985

Ford G.A., Wiener R., 'Modula-2, A Software Development Approach', Wiley 1985

Wirth N., 'Programming in Modula-2', derde verbeterde druk, Springer-Verlag 1985

Oefeningen

Beschouw de declaratie

```
TYPE Knooppunt = RECORD
       waarde : CARDINAL;
       wijzer : Wijzer
     END;
     Wijzer   = POINTER TO Knooppunt;
```

1. Schrijf een functieprocedure die uit een lijst de waarde van het k-de element oplevert.

2. Schrijf een functieprocedure voor het berekenen van het aantal elementen van een lijst.

3. Schrijf een functieprocedure voor het kopiëren van een lijst. Het functie-argument is de kop van de originele lijst, het functieresultaat is de kop van de nieuwe lijst.

4. De functieprocedure voegt een kopie van lijst y toe aan het einde van lijst x. Het functieresultaat is de kop van de nieuwe lijst.

5. Formuleer recursieve oplossingen voor de opgaven 1 - 4.

6. Schrijf de procedure Map die op elk lijstelement de monadische functie F toepast. Het type van F wordt gegeven door :

```
TYPE MapFunctie = PROCEDURE(CARDINAL) : CARDINAL;
```

7. De invoer voor de functieprocedure DyadischFunctie bestaat uit de lijsten x en y en de functieprocedure f met type

```
TYPE Functie = PROCEDURE(CARDINAL, CARDINAL) : CARDINAL;
```

De lijsten x en y hebben een gelijke lengte. De functieprocedure creëert een nieuwe lijst door het elementsgewijs toepassen van functie f op de lijsten x en y. Het functieresultaat is de kop van de nieuwe lijst. Schrijf de procedure DyadischeFunctie.

8. Schrijf procedure VoegNa voor het toevoegen van een element na een knooppunt met een gegeven waarde.

9. Schrijf procedure VoegVoor voor het toevoegen van een element vóór een knooppunt met een gegeven waarde.

1Ø. Definieer een gegevensstructuur en procedures voor het beheer van een dynamische matrix met een willekeurig aantal dimensies.

9 Bibliotheekmodulen

Programmatuur wordt reeds geruime tijd ontwikkeld volgens de methode van gestructureerd programmeren. Een programma kan dan zo worden opgebouwd dat elke belangrijke actie in de formulering van het algoritme overeenstemt met de aanroep van een procedure. Deze procedures worden, bijvoorbeeld in een Pascal-programma, samen met alle andere declaraties gegroepeerd aan het begin van de programmatekst. In de vorige hoofdstukken is reeds geschreven dat we in een Modula-2 programma de objecten die logisch bij elkaar horen ook in de programmatekst kunnen groeperen. Deze groepering wordt in Modula-2 actief ondersteunt door het modulemechanisme.

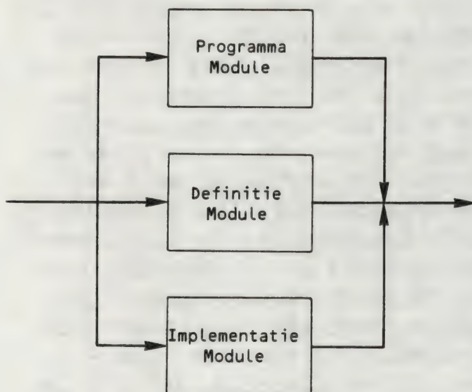
Een module is een afzonderlijk stuk programma dat autonoom kan worden gedefinieerd en waarin een verzameling declaraties en procedures logisch worden gegroepeerd. Een module kapselt een deel van een programma in. We onderscheiden vier typen modulen : een programmamodule, twee typen bibliotheekmodulen en een interne module. Een programmamodule is een volledig programma dat, over het algemeen, objecten gebruikt die in bibliotheekmodulen zijn gedefinieerd. Een bibliotheekmodule bestaat uit twee delen : de specificatie van de module en de realisatie ervan. We specificeren de bibliotheekmodule met een definitiemodule, DEFINITION MODULE. We implementeren de bewerkingen en de gegevensstructuren met de implementatiemodule, IMPLEMENTATION MODULE. Een bibliotheek bevat over het algemeen programmatuurcomponenten die we dikwijls gebruiken. Een gebruikersmodule is een module waarin objecten worden gebruikt die in een bibliotheek zijn gedefinieerd. De gebruikersmodule voert deze objecten in (IMPORT), de bibliotheekmodule voert ze uit (EXPORT). Een interne module kan worden gedefinieerd in de romp van een programma- of een implementatiemodule.

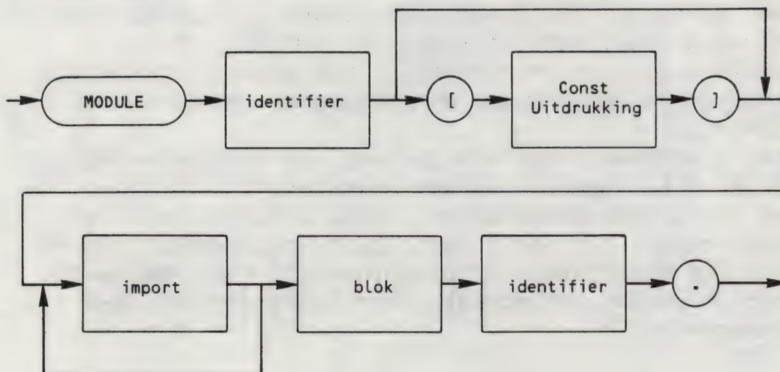
Elk Modula-2 systeem bevat een verzameling bibliotheekmodulen voor het gebruik en het beheer van het computersysteem en voor de uitvoering van specifieke bewerkingen. Deze bibliotheek is nog niet gestandaardiseerd. In een volgend hoofdstuk beschrijven we een bibliotheek die als standaard is voorgesteld door Modula-2 gebruikers en door enkele belangrijke bedrijven, die Modula-2 systemen implementeren.

9.1 Modulen compileren

Alle modulen, behalve de interne modulen, kunnen afzonderlijk worden gecompileerd. Een autonome module, die de invoer vormt voor de Modula-2 compiler, wordt een compileereenheid genoemd. De syntaxis voor een compileereenheid is :

```
ProgrammaModule  = 'MODULE' identifier ';'
                   { import }
                   blok
                   identifier '.'
CompileerEenheid = DefinitieModule
                 | [ 'IMPLEMENTATION' ] ProgrammaModule
```



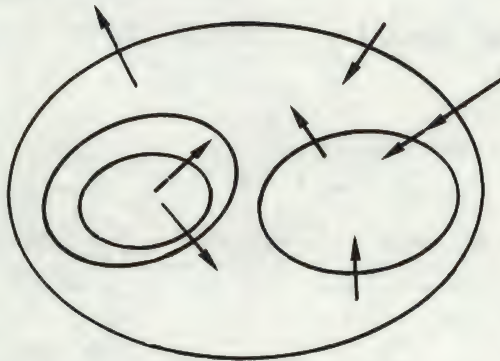


De Modula-2 omgeving ondersteunt de overdracht van objecten tussen de afzonderlijk gecompileerde modulen : een gebruikersmodule bevat externe verwijzingen naar de objecten die in een bibliotheekmodule zijn gedefinieerd. Met deze externe verwijzingen worden de volgende controles op de overdraagbaarheid van de objecten uitgevoerd :

- voor alle objecten die een gebruikersmodule invoert moeten de externe verwijzingen bestaan in gecompileerde vorm;
- voor elke externe aanroep van een procedure wordt de actuele parameterlijst vergeleken met de formele parameterlijst in de bibliotheekmodule. De controles omvatten : het aantal parameters, het type van elke parameter en de parameterbinding;
- bij de compilatie wordt aan elke module een versienummer toegekend. Het versienummer van de implementatie wordt vergeleken met het versienummer van de definitiemodulen. Bij elke nieuwe compilatie van een definitiemodule wijzigt ook het versienummer. Alle gebruikersmodulen moeten dan opnieuw worden gecompileerd. Hierdoor wordt voorkomen dat in een systeem verschillende gebruikersmodulen gebaseerd zijn op verschillende versies van dezelfde definitiemodule.

9.2 In- en uitvoer van objecten

We kunnen een module beschouwen als een logische eenheid die door een membraan is omgeven.

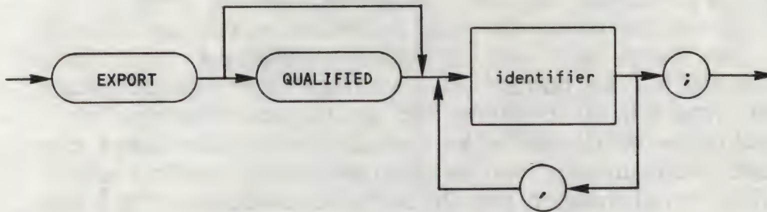


figuur 9.1 IMPORT en EXPORT van objecten tussen modulen

De informatie wordt van de omgeving door het membraan heen in de module ingevoerd met een invoerlijst, en van de module naar de omgeving met een uitvoerlijst. Een gebruikersmodule bevat één of meer invoerlijsten, een bibliotheekmodule bevat eventueel één of meer invoerlijsten en slechts één uitvoerlijst.

De syntaxis voor de uitvoerlijst is :

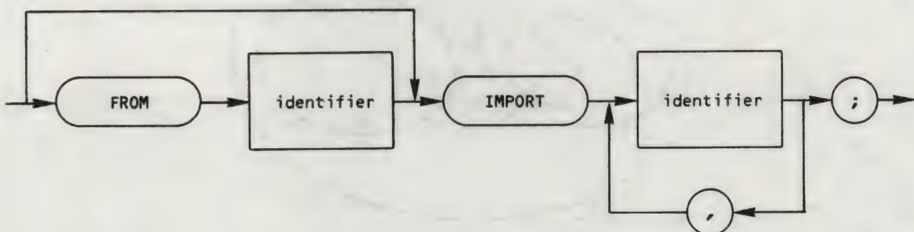
export = 'EXPORT' ['QUALIFIED'] IdentLijst ';'



De IdentLijst bevat de opsomming van alle objecten die de bibliotheekmodule uitvoert. Volgens de derde druk van 'Programming in Modula-2' van Wirth (1985) worden alle objecten van een definitiemodule automatisch uitgevoerd. De uitvoerlijst is hier overbodig. Met een uitvoerlijst wordt door een Modula-2 compiler, die volgens deze laatste specificaties is geïmplementeerd, geen rekening gehouden. Oudere bibliotheekmodulen zijn hierdoor overdraagbaar naar recente compilers.

De syntaxis voor een invoerlijst is :

import = ['FROM' identifier] 'IMPORT' IdentLijst ';'



De identifier na het FROM-symbool is de naam van de module waaruit objecten worden ingevoerd. Het FROM-symbool gevolgd door een modulenaam is facultatief. De betekenis van de IdentLijst is afhankelijk van de aanwezigheid van de FROM-clausule :

met de clausule bevat de IdentLijst een opsomming van alle objecten die we uit de module 'identifier' invoeren.

Voorbeeld :
```
FROM SimpleIO IMPORT ReadCard, ReadString;
FROM Storage  IMPORT ALLOCATE, DEALLOCATE;
```

Zonder de FROM-clausule is de IdentLijst slechts een opsomming van de identifiers van de modulen die we willen invoeren.

Voorbeeld :
```
IMPORT SimpleIO, Storage;
```

De betekenis van de FROM-clausule is vergelijkbaar met de betekenis van de WITH-opdracht : met de WITH-opdracht definiëren we een bereik in de tekst waar de namen van recordvelden als variabelen kunnen worden gebruikt. Met de FROM-clausule kunnen we naar alle objecten van de lijst verwijzen zoals naar objecten die in de gebruikersmodule zelf zijn gedefinieerd. Zonder deze clausule voeren we alleen de modulenaam in. Alle objecten die deze module uitvoert worden automatisch mee ingevoerd. Dit impliceert echter niet dat de objecten van deze module automatisch zichtbaar worden in de gebruikersmodule. De objecten zijn alleen toegankelijk in de gebruikersmodule met een gekwalificeerde identifier. De gekwalificeerde identifier is samengesteld uit de identifier van de module, een punt en de identifier van het object.

We gebruiken een importlijst zonder de FROM-clausule als we objecten met dezelfde identifier uit verschillende modulen invoeren. Met de gekwalificeerde verwijzing kunnen deze objecten in de gebruikersmodule van elkaar worden onderscheiden. Ook wordt de leesbaarheid van een programma verbeterd door het gebruik van gekwalificeerde identifiers.

Voorbeeld :
```
FROM Files    IMPORT File;
FROM Terminal IMPORT WriteString;
IMPORT SimpleIO;
IMPORT Text;

VAR a, b, c     : ARRAY [0..20] OF CHAR;
    f           : File;

WriteString(a);

SimpleIO.WriteString(b);
Text.WriteString(f,c);
```

Impliciete in- en uitvoer

Als een recordtype uit een module wordt uitgevoerd (EXPORT), worden automatisch ook alle identifiers van de recordvelden uitgevoerd. Deze regel geldt ook voor de constante identifiers die met een enumeratietype worden gedeclareerd. Op een analoge wijze impliceert de invoer van een identifier die als een recordtype of als een enumeratietype is gedeclareerd, automatisch ook de invoer van de identifiers van de recordvelden of van de constante identifiers.

Voorbeeld :

In de definitiemodule Tijd definiëren we de uitvoerlijst :

```
EXPORT QUALIFIED Maand, Datum;
```

met

```
TYPE Maand = (januari, februari, maart, april, mei, juni,
              juli, augustus, september, oktober, november,
              december);
     Datum = RECORD
       dag     : [1..31];
       maand   : Maand;
       jaar    : [1900..1999]
     END;
```

en in een gebruikersmodule M de invoerlijst :

```
FROM Tijd IMPORT Datum, Maand;
```

en de declaraties

```
VAR m  : Maand;
    d  : Datum;
```

Dan zijn in module M de volgende verwijzingen naar de constante identifiers met type Maand of naar de velden van een record met type Datum mogelijk :

```
m := november;
d.dag := 23;
d.maand := m;
d.jaar := 1986;
```

De in- en uitvoer van procedures met formele parameters

Bij de uitvoer van een procedure, die in een definitiemodule is gedefinieeerd, worden de typen van de formele parameters niet automatisch mee uitgevoerd. Een gebruikersmodule moet daarom zowel de procedure als de typen van de formele parameters invoeren.

Voorbeeld :
In de bibliotheekmodule StandardIO wordt de procedure SetEchoMode gedefinieerd. Met deze procedure wordt de automatische weergave van de standaardinvoer naar de standaarduitvoer ingesteld. De procedure SetEchoMode heeft één parameter met type EchoMode :

```
TYPE EchoMode = (echo, noEcho);
```

De procedurekop van SetEchoMode is dan :

```
PROCEDURE SetEchoMode(echo : EchoMode);
```

De gebruikersmodule waarin de echo moet worden ingesteld voert de objecten SetEchoMode en EchoMode in met de invoerlijst

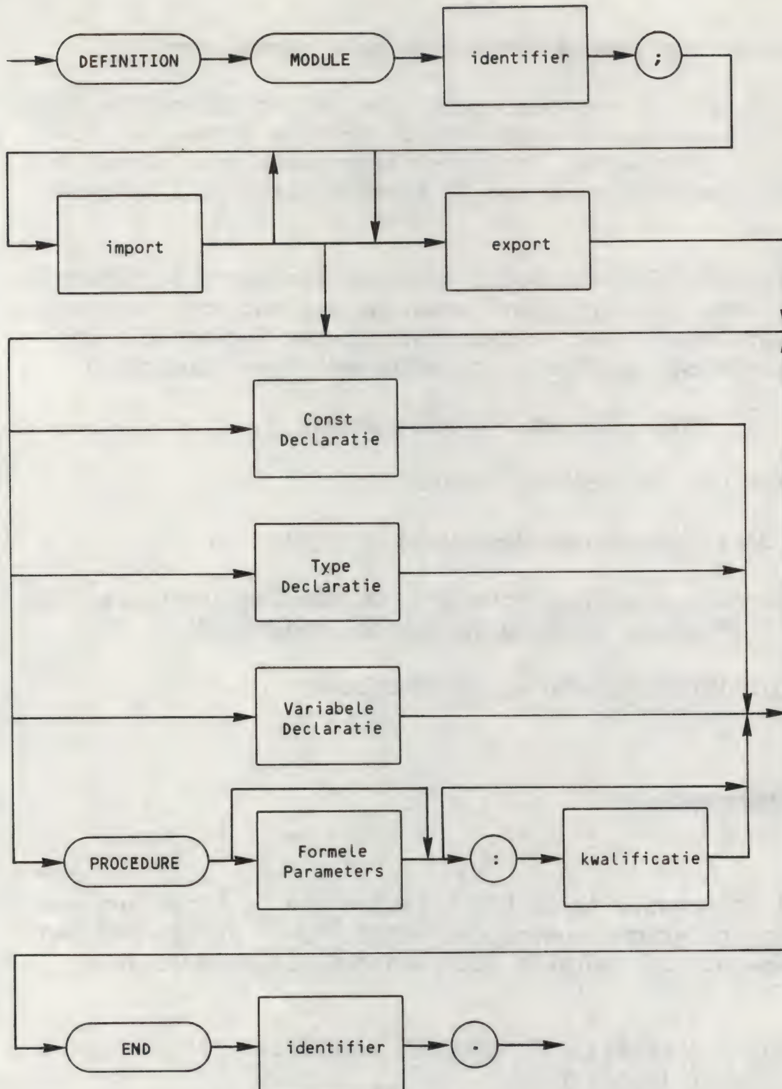
```
FROM StandardIO IMPORT EchoMode, SetEchoMode;
```

9.3 Definitiemodule

Een definitiemodule is de beschrijving van de koppeling van een verzameling programmatuurcomponenten die naar andere modulen worden uitgevoerd. De syntaxis voor een definitiemodule is :

```
DefinitieModule = 'DEFINITION' 'MODULE' identifier ';'
                  { import }
                  [ export ]
                  { definitie }
                  'END' identifier '.'

definitie       =   'CONST' { ConstDeclaratie ';' }
                  | 'TYPE'  { identifier [ '=' type ] ';' }
                  | 'VAR'   { VariabeleDeclaratie ';' }
                  | ProcedureKop ';'
```



In een definitiemodule kunnen geen of één of meer invoerlijsten voorkomen. De uitvoerlijst bevat de opsomming van alle objecten die worden uitgevoerd. Tot deze lijst kunnen behoren : constanten, typen, variabelen en procedures. Volgens de recente Modula-2 specificaties is de uitvoerlijst overbodig; alle objecten die in de definitiemodule worden gedefinieerd worden automatisch uitgevoerd. Na de in- en uitvoerlijsten volgt een opsomming van declaraties. De constanten en de variabelen worden zoals gewoonlijk gedeclareerd. Een type kunnen we op twee manieren specificeren : als verborgen type (Engels : opaque type; hidden type) of als transparant type. We leggen dit verschil in de volgende paragraaf uit. Voor de procedures wordt alleen de procedurekop gedecla-

reerd. De procedurekop bevat de naam van de procedure, de formele parameterlijst en het resultaattype ingeval het een functieprocedure betreft. De procedures worden geïmplementeerd in een implementatiemodule.

Transparante en verborgen typen

Een type in een definitiemodule is transparant voor de gebruikersmodulen indien de gegevensstructuur van het type in de definitiemodule bekend is. De gebruikersmodulen van de definitiemodule kunnen deze gegevensstructuur invoeren en aldus bewerkingen op de interne objecten van de structuur uitvoeren. We illustreren dit met een definitiemodule voor de bewerkingen met een dynamische matrix.

```
DEFINITION MODULE DynMat;
EXPORT QUALIFIED
  (* type *) Matrix, DimensieIndex,
  (* proc *) CreeerMatrix, GeefElementVrij, Hoog, Laag,
             WaardeElement, SchrijfElement;
TYPE DimensieIndex = [1..2];
     Matrix   = RECORD
       rijOndergrens   : INTEGER;
       rijBovengrens   : INTEGER;
       kolomOndergrens : INTEGER;
       kolomBovengrens : INTEGER;
       element         : POINTER TO CARDINAL
      END;

PROCEDURE CreeerMatrix(    rijOnder   : INTEGER;
                           rijBoven   : INTEGER;
                           kolomOnder : INTEGER;
                           kolomBoven : INTEGER;
                       VAR m          : Matrix);

PROCEDURE GeefElementVrij(VAR m : Matrix);

PROCEDURE Hoog(    m : Matrix;
                   k : DimensieIndex) : INTEGER;

PROCEDURE Laag(    m : Matrix;
                   k : DimensieIndex) : INTEGER;

PROCEDURE SchrijfElement(     m      : Matrix;
                              i      : INTEGER;
                              j      : INTEGER;
                              waarde : CARDINAL);
```

```
PROCEDURE WaardeElement(      m        : Matrix;
                              i        : INTEGER;
                              j        : INTEGER) : CARDINAL;

END DynMat.
```

Het type Matrix is transparant : de gegevensstructuur is volledig bekend in de definitiemodule. Module M voert de objecten van DynMat in en kan op de elementen van de structuur met type Matrix bewerkingen uitvoeren :

```
MODULE M;
FROM DynMat IMPORT Matrix, CreeerMatrix;

VAR a : Matrix;

BEGIN

...
a.rijOndergrens := -10;
a.element  := NIL;
...
END M.
```

Bij veel toepassingen is het niet gewenst dat de gebruikersmodulen een gegevensstructuur rechtstreeks manipuleren. Met een verborgen type wordt dit verhinderd : in de definitiemodule wordt alleen de naam van het type gedeclareerd. We definiëren zo'n abstract gegevenstype. Alle details betreffende de gegevensstructuur worden geïmplementeerd in de implementatiemodule en zijn ontoegankelijk voor de gebruikersmodulen. Het gebruik van een verborgen type is beperkt tot de wijzertypen. Een wijzertype is steeds gebonden aan een ander type waarvan de structuur kan worden verborgen. We definiëren opnieuw de definitiemodule DynMat maar nu met een verborgen matrixtype :

```
DEFINITION MODULE DynMat;
EXPORT QUALIFIED
  (* type *) Matrix, DimensieIndex,
  (* proc *) CreeerMatrix, WisMatrix, Hoog, Laag,
             WaardeElement, SchrijfElement;

TYPE DimensieIndex = [1..2];
     Matrix;
```

```
PROCEDURE CreeerMatrix(    rijOnder   : INTEGER;
                           rijBoven   : INTEGER;
                           kolomOnder : INTEGER;
                           kolomBoven : INTEGER;
                       VAR m          : Matrix);

PROCEDURE WisMatrix(VAR m : Matrix);

PROCEDURE Hoog(    m : Matrix;
                   k : DimensieIndex) : INTEGER;

PROCEDURE Laag(    m : Matrix;
                   k : DimensieIndex) : INTEGER;

PROCEDURE SchrijfElement(  m          : Matrix;
                           i          : INTEGER;
                           j          : INTEGER;
                           waarde     : CARDINAL);

PROCEDURE WaardeElement(   m          : Matrix;
                           i          : INTEGER;
                           j          : INTEGER) : CARDINAL;
END DynMat.
```

Het type Matrix is een verborgen type : de gegevensstructuur is onbekend in de definitiemodule. Module M voert de objecten van DynMat in, maar kan op de elementen van de structuur met type Matrix geen bewerkingen uitvoeren :

```
MODULE M;
FROM DynMat IMPORT Matrix, CreeerMatrix;
VAR a : Matrix;

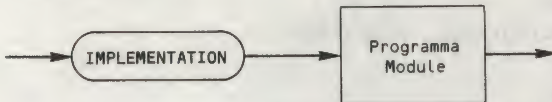
BEGIN
CreeerMatrix(1, 10, 1, 10, a);
...
END M.
```

9.4 Implementatiemodule

De implementatiemodule bevat alle implementatiedetails voor de specificaties van de definitiemodule : de volledige declaraties van de procedures en de definities van de gegevensstructuren met een verborgen type. De implementatiemodule heeft dezelfde vorm als een programmamodule.

De syntaxis voor een implementatiemodule is :

ImplementatieModule = 'IMPLEMENTATION' ProgrammaModule



Alle objecten die in de definitiemodule worden gedefinieerd, worden automatisch in de implementatiemodule ingevoerd. Alle objecten die in de definitiemodule worden ingevoerd, zijn echter niet automatisch toegankelijk in de implementatiemodule. Voor deze objecten moeten eventueel nieuwe invoerlijsten in de implementatiemodule worden gespecificeerd. In een implementatiemodule kunnen we ook constanten, typen, variabelen en procedures definiëren die niet in de definitiemodule worden gespecificeerd. Deze objecten zijn lokaal en zijn dus onzichtbaar voor modulen buiten de implementatiemodule.

Als voorbeeld implementeren we nu het transparante type Matrix met zijn bewerkingen :

```
IMPLEMENTATION MODULE DynMat;
FROM SYSTEM      IMPORT TSIZE, ADDRESS;
FROM Storage     IMPORT ALLOCATE, DEALLOCATE;

PROCEDURE CreeerMatrix(     rijOnder   : INTEGER;
                            rijBoven   : INTEGER;
                            kolomOnder : INTEGER;
                            kolomBoven : INTEGER;
                        VAR m          : Matrix);

BEGIN
WITH m DO
  rijOndergrens := rijOnder;
  rijBovengrens := rijBoven;
  kolomOndergrens := kolomOnder;
  kolomBovengrens := kolomBoven;
  ALLOCATE(element,
    TSIZE(CARDINAL) * CARDINAL((Hoog(m, 1) - Laag(m, 1) + 1) *
                               (Hoog(m, 2) - Laag(m, 2) + 1)))
END
END CreeerMatrix;
```

```
PROCEDURE GeefElementVrij(VAR m : Matrix);
BEGIN
WITH m DO
  DEALLOCATE(element,
    TSIZE(CARDINAL) * CARDINAL((Hoog(m, 1) - Laag(m, 1) + 1) *
                               (Hoog(m, 2) - Laag(m, 2) + 1)))
END
END GeefElementVrij;


PROCEDURE Hoog(    m : Matrix;
                   k : DimensieIndex) : INTEGER;
BEGIN
IF k = 1
THEN
  RETURN m.rijBovengrens
ELSE
  RETURN m.kolomBovengrens
END
END Hoog;


PROCEDURE Laag(    m : Matrix;
                   k : DimensieIndex) : INTEGER;
BEGIN
IF k = 1
THEN
  RETURN m.rijOndergrens
ELSE
  RETURN m.kolomOndergrens
END
END Laag;


PROCEDURE Verplaatsing(    m     : Matrix;
                           rij   : INTEGER;
                           kolom : INTEGER) : CARDINAL;
BEGIN
RETURN CARDINAL(
        (rij - Laag(m, 1)) * (Hoog(m, 2) - Laag(m, 2) + 1) +
                kolom - Laag(m, 2) + 1)
       * TSIZE(CARDINAL)
END Verplaatsing;

PROCEDURE SchrijfElement(    m           : Matrix;
                             i           : INTEGER;
                             j           : INTEGER;
                             waarde      : CARDINAL);
VAR adres : POINTER TO CARDINAL;
```

```
BEGIN
adres := ADDRESS(m.element) + Verplaatsing(m, i, j);
adres↑ := waarde
END SchrijfElement;


PROCEDURE WaardeElement(   m          : Matrix;
                           i          : INTEGER;
                           j          : INTEGER) : CARDINAL;
VAR adres : POINTER TO CARDINAL;

BEGIN
adres := ADDRESS(m.element) + Verplaatsing(m,i,j);
RETURN adres↑
END WaardeElement;

END DynMat.
```

In het volgende voorbeeld implementeren we het verborgen type Matrix met zijn bewerkingen. Een verborgen type is steeds een wijzertype en bij de implementatie moeten we hiervoor de nodige voorzieningen treffen : de import van de procedures ALLOCATE en DEALLOCATE, de toewijzing van de geheugenruimte aan het object, de adressering van de waarde met behulp van de dereferentieoperator.

```
IMPLEMENTATION MODULE DynMat;
FROM SYSTEM     IMPORT TSIZE, ADDRESS;
FROM Storage    IMPORT ALLOCATE, DEALLOCATE;

(* definitie van het verborgen type Matrix *)
TYPE Matrix = POINTER TO RECORD
       rijOndergrens    : INTEGER;
       rijBovengrens    : INTEGER;
       kolomOndergrens  : INTEGER;
       kolomBovengrens  : INTEGER;
       element          : POINTER TO CARDINAL
     END;

PROCEDURE CreeerMatrix(    rijOnder   : INTEGER;
                           rijBoven   : INTEGER;
                           kolomOnder : INTEGER;
                           kolomBoven : INTEGER;
                       VAR m          : Matrix);
```

```
BEGIN
NEW(m);
WITH m↑ DO
  rijOndergrens := rijOnder;
  rijBovengrens := rijBoven;
  kolomOndergrens := kolomOnder;
  kolomBovengrens := kolomBoven;
  ALLOCATE(element,
    TSIZE(CARDINAL) * CARDINAL((Hoog(m, 1) - Laag(m, 1) + 1) *
                               (Hoog(m, 2) - Laag(m, 2) + 1)))
END
END CreeerMatrix;


PROCEDURE WisMatrix(VAR m : Matrix);
BEGIN
WITH m↑ DO
  DEALLOCATE(element,
    TSIZE(CARDINAL) * CARDINAL((Hoog(m, 1) - Laag(m, 1) + 1) *
                               (Hoog(m, 2) - Laag(m, 2) + 1)))
END;
DISPOSE(m)
END WisMatrix;


PROCEDURE Hoog(     m : Matrix;
                    k : DimensieIndex) : INTEGER;
BEGIN
IF k = 1
THEN
  RETURN m↑.rijBovengrens
ELSE
  RETURN m↑.kolomBovengrens
END
END Hoog;


PROCEDURE Laag(     m : Matrix;
                    k : DimensieIndex) : INTEGER;
BEGIN
IF k = 1
THEN
  RETURN m↑.rijOndergrens
ELSE
  RETURN m↑.kolomOndergrens
END
END Laag;
```

```
PROCEDURE Verplaatsing(    m     : Matrix;
                           rij   : INTEGER;
                           kolom : INTEGER) : CARDINAL;
BEGIN
RETURN CARDINAL(
        (rij - Laag(m, 1)) * (Hoog(m, 2) - Laag(m, 2) + 1) +
               kolom - Laag(m, 2) + 1)
      * TSIZE(CARDINAL)
END Verplaatsing;


PROCEDURE SchrijfElement(  m           : Matrix;
                           i           : INTEGER;
                           j           : INTEGER;
                           waarde      : CARDINAL);
VAR adres : POINTER TO CARDINAL;

BEGIN
adres := ADDRESS(m↑.element) + Verplaatsing(m, i, j);
adres↑ := waarde
END SchrijfElement;


PROCEDURE WaardeElement(   m           : Matrix;
                           i           : INTEGER;
                           j           : INTEGER) : CARDINAL;
VAR adres : POINTER TO CARDINAL;

BEGIN
adres := ADDRESS(m↑.element) + Verplaatsing(m,i,j);
RETURN adres↑
END WaardeElement;

END DynMat.
```

Voor het verborgen type Matrix definiëren we nu nog een bibliotheekmodule MatIO voor de in- en uitvoer van de matrixelementen. We beschouwen eerst de definitiemodule :

```
DEFINITION MODULE MatIO;
FROM DynMat IMPORT Matrix;
EXPORT QUALIFIED
  (* type *) InProcedure, UitProcedure,
  (* proc *) LeesTabel, DrukTabel;

TYPE InProcedure  = PROCEDURE (VAR CARDINAL, VAR BOOLEAN);
     UitProcedure = PROCEDURE (CARDINAL, VAR BOOLEAN);
```

```
PROCEDURE LeesTabel(     m       : Matrix;
                    VAR succes : BOOLEAN;
                         in      : InProcedure);
PROCEDURE DrukTabel(     m       : Matrix;
                    VAR succes : BOOLEAN;
                         uit     : UitProcedure);
END MatIO.
```

De procedure LeesTabel leest de elementen van een tabel rij voor rij, de procedure DrukTabel drukt de elementen rij voor rij af. Als de in- of uitvoer van de tabel geheel succesvol is afgewerkt, wordt de waarde van de parameter succes gelijk aan TRUE. Anders is de waarde van succes gelijk aan FALSE. De oorsprong van de elementen, de vorm van de in- en uitvoer en het resultaat van elke lees- of schrijfopdracht, worden bepaald door de actuele waarden van de procedureparameters in en uit. Met behulp van de proceduretypen InProcedure en UitProcedure, definiëren we in een gebruikersmodule de procedures voor de in- en uitvoer van elk afzonderlijk element. Deze procedures worden als een parameter aan de procedures LeesTabel en DrukTabel doorgegeven.

Voorbeeld :
In een bepaalde toepassing worden de elementen van een tabel gelezen van de standaardinvoer en afgedrukt op de standaarduitvoer. Het standaardinvoerbestand bevat telkens één element per regel. Op de standaarduitvoer worden alle elementen op dezelfde regel afgedrukt. Voor elk element voorzien we vijf afdrukposities. We definiëren hiervoor de procedures LeesElement en DrukElement :

```
IMPORT SimpleIO;
PROCEDURE LeesElement(VAR element : CARDINAL;
                      VAR succes  : BOOLEAN);
BEGIN
SimpleIO.ReadCard(element, succes);
SimpleIO.ReadLn
END LeesElement;

PROCEDURE SchrijfElement(    element : CARDINAL;
                         VAR succes  : BOOLEAN);
BEGIN
SimpleIO.WriteCard(element, 5);
succes := TRUE
END SchrijfElement;
```

Een tabel a kan nu worden gelezen en afgedrukt met de opdrachten :

```
LeesTabel(a, ok, LeesElement);
DrukTabel(a, ok, SchrijfElement)
```

We besluiten met de implementatie van de bibliotheekmodule MatIO :

```
IMPLEMENTATION MODULE MatIO;
FROM DynMat  IMPORT Matrix,
                    Laag, Hoog,
                    SchrijfElement, WaardeElement;

PROCEDURE LeesTabel(    m      : Matrix;
                    VAR succes : BOOLEAN;
                        in     : InProcedure);
(* inlezen van de elementen van een dynamische matrix met
   basistype CARDINAL *)
VAR i, j    : INTEGER;
    element : CARDINAL;

BEGIN
FOR i := Laag(m, 1) TO Hoog(m, 1) DO
  FOR j := Laag(m, 2) TO Hoog(m, 2) DO
    in(element, succes);
    IF NOT succes
    THEN
      RETURN
    END;
    SchrijfElement(m, i, j, element)
  END
END
END LeesTabel;

PROCEDURE DrukTabel(    m      : Matrix;
                    VAR succes : BOOLEAN;
                        uit    : UitProcedure);
(* afdrukken van de elementen van een dynamische matrix met
   basistype CARDINAL *)
VAR i, j    : INTEGER;
    element : CARDINAL;

BEGIN
FOR i := Laag(m, 1) TO Hoog(m, 1) DO
  FOR j := Laag(m, 2) TO Hoog(m, 2) DO
    uit(WaardeElement(m, i, j), succes);
    IF NOT succes
    THEN
      RETURN
    END
  END
END
END DrukTabel;
END MatIO.
```

9.5 Het initiëren van variabelen in een implementatiemodule

In een implementatiemodule kunnen we ook een aantal opdrachten specificeren die automatisch worden uitgevoerd voordat de uitvoering van de gebruikersmodule start. Met deze opdrachten initiëren we de gegevensstructuren van een bibliotheekmodule. De vorm van de implementatiemodule stemt overeen met een programmamodule, de opdrachten voor de initiëring stemmen overeen met de romp van een procedure. Deze procedure kan echter niet expliciet worden aangeroepen.

Voorbeeld :
We ontwerpen een programma dat de frequentie van elke kleine letter in een bestand telt. Voor het bijhouden van de letterfrequenties definiëren we een bibliotheekmodule met de bewerkingen TelFrequentie en Frequentie. Voor deze toepassing gebruiken we slechts één frequentietabel. De gegevensstructuur hiervoor wordt in de implementatiemodule gedeclareerd.

```
DEFINITION MODULE FreqTab;
EXPORT QUALIFIED
  (* proc *) TelFrequentie, Frequentie;

PROCEDURE TelFrequentie(teken : CHAR);
PROCEDURE Frequentie(teken : CHAR) : CARDINAL;

END FreqTab.
```

```
IMPLEMENTATION MODULE FreqTab;
TYPE Index = ['a'..'z'];
     FrequentieTabel = ARRAY Index OF CARDINAL;
VAR  tabel : FrequentieTabel;

PROCEDURE TelFrequentie(teken : CHAR);

BEGIN
INC(tabel[teken])
END TelFrequentie;


PROCEDURE Frequentie(teken : CHAR) : CARDINAL;

BEGIN
RETURN tabel[teken]
END Frequentie;
```

```
(* initiëring van de frequentietabel *)
VAR i : Index;

BEGIN
FOR i := 'a' TO 'z' DO
  tabel[i] := 0
END
END FreqTab.
```

Een module M gebruikt deze bibliotheekmodule. De tabel wordt automatisch geïnitieerd voordat de opdrachten van module M worden gestart.

```
MODULE M;
FROM FreqTab IMPORT TelFrequentie, Frequentie;
BEGIN
...
END M.
```

Als een module een gebruiker is van verschillende bibliotheekmodulen waarin telkens initiëringsopdrachten voorkomen, worden deze initiëringen uitgevoerd in de volgorde van de invoerlijsten.

Voorbeeld :

```
IMPLEMENTATION MODULE A;
FROM SimpleIO IMPORT WriteString, WriteLn;
...
BEGIN
WriteString('In module A');
WriteLn
END A.


IMPLEMENTATION MODULE B;
FROM SimpleIO IMPORT WriteString, WriteLn;
...
BEGIN
WriteString('In module B');
WriteLn
END B.


IMPLEMENTATION MODULE C;
FROM SimpleIO IMPORT WriteString, WriteLn;
...
```

```
BEGIN
WriteString('In module C');
WriteLn
END C.

MODULE M;
IMPORT A;
IMPORT C;
IMPORT B;
BEGIN
...
END M.
```

De module M voert de bibliotheekmodulen A, C en B in. De initiëringsopdrachten van deze modulen worden in dezelfde volgorde uitgevoerd als van de importlijsten. Op de standaarduitvoer worden dus de volgende teksten afgedrukt :

```
In module A
In module C
In module B
```

9.6 De levensduur van variabelen in een implementatiemodule

Beschouw de declaratie van de procedure P :

```
PROCEDURE P;
VAR a, b : CARDINAL;

BEGIN
...
END P;
```

De procedure P definieert het bereik van de variabelen a en b in de programmatekst. Zij zijn lokaal voor de procedure P. Deze variabelen zijn ook dynamisch : bij de aanroep van P worden a en b gecreëerd, bij de beëindiging van P wordt de geheugenruimte voor a en b weer vrijgegeven. De levensduur van deze lokale variabelen is zo lang als de procedure actief is.

Beschouw nu opnieuw de implementatiemodule FreqTab. In deze module hebben we de tabelvariabele met type FrequentieTabel gedefinieerd. Ook de variabele tabel is lokaal : deze variabele is alleen toegankelijk in de implementatiemodule FreqTab. De variabele tabel is echter statisch : de levensduur is gelijk aan de levensduur van het programma dat de module invoert.

Met de variabelen die in een implementatiemodule worden gedeclareerd beschikken we nu over een mechanisme waarbij de levensduur van de variabelen overeenstemt met de levensduur van het programma en waarbij het bereik van de variabelen goed wordt afgebakend door de omhullende module. In andere programmeertalen moeten we hiervoor globale variabelen gebruiken met alle daaraan verbonden nadelen : globale variabelen zijn overal toegankelijk zodat ongeoorloofde bewerkingen op deze variabelen kunnen worden uitgevoerd en de leesbaarheid van de programmatekst sterk wordt verminderd.

9.7 De volgorde bij het compileren

De Modula-2 omgeving ondersteunt de uitwisseling van objecten zodat een module objecten kan gebruiken die door een bibliotheekmodule worden uitgevoerd. De meeste Modula-2 implementaties gebruiken hiervoor het volgende mechanisme : bij de compilatie van een definitiemodule wordt een symbolenbestand aangemaakt. Dit symbolenbestand bevat een samenvatting van de declaraties uit de definitiemodule en ook een versienummer. Dit symbolenbestand wordt later gebruikt voor het uitvoeren van controles op de typen en op de procedures bij de invoer van objecten in de gebruikersmodulen. Ook voor de compilatie van de implementatiemodule wordt dit bestand gebruikt. In beide gevallen wordt het versienummer gekopieerd in de codebestanden van de desbetreffende modulen. Bij het laden of het koppelen van de modulen wordt de consistentie van de verschillende versienummers nagegaan. Bij eventuele fouten moeten de modulen met een foutief versienummer opnieuw worden gecompileerd.

We beschreven deze versiecontrole reeds in het hoofdstuk 'De Modula-2 ontwikkelingsomgeving'. Voor de duidelijkheid illustreren we de versiecontrole nog eens met een voorbeeld.

Beschouw de definitiemodule A :

```
DEFINITION MODULE A;
EXPORT QUALIFIED P;

PROCEDURE P(x, y : INTEGER);
END A.
```

Na de compilatie van deze module bevat het symbolenbestand een versienummer, bijvoorbeeld v_1.

```
MODULE B;
FROM A IMPORT P;
```

```
(* declaraties B *)

BEGIN
...
P(23, 4);
...
END B.
```

Na de compilatie van module B bevat het codebestand het versienummer v_1. De compiler kopieert dit versienummer uit het symbolenbestand van module A.

```
IMPLEMENTATION MODULE A;

PROCEDURE P(x, y : INTEGER);
BEGIN
(* opdrachten voor P *)
END P;

BEGIN
(* initiëring A *)
END A.
```

Het codebestand voor de implementatiemodule bevat het versienummer v_1. De compiler kopieert dit versienummer uit het symbolenbestand van de definitiemodule A. De implementatiemodule mogen we opnieuw wijzigen en compileren. Zolang de implementatie beantwoordt aan de specificaties van de definitiemodule behoeven we module B niet te hercompileren.
We wijzigen nu de specificaties van module A :

```
DEFINITION MODULE A;
EXPORT QUALIFIED P;

PROCEDURE P(x, y : CARDINAL);
END A.
```

Na de compilatie van deze module bevat het symbolenbestand het versienummer v_2. Zowel de implementatiemodule van A als module B moeten opnieuw worden gecompileerd zodat in de codebestanden het versienummer v_2 wordt geregistreerd. Eventueel moet eerst de programmatekst worden aangepast aan de nieuwe specificaties.

9.8 Het ontwerp van bibliotheekmodulen

In deze paragraaf geven we enkele richtlijnen voor het ontwerp van modulen :

- Beperk de invoer van objecten in een definitiemodule.

  Elke nieuwe compilatie van een definitiemodule veroorzaakt een kettingreactie van uit te voeren compilaties om de consistentie van de versies in stand te houden. De omvang van deze kettingreactie is afhankelijk van het aantal objecten dat in elke afhankelijke definitiemodule wordt ingevoerd.

- Vermijd de wederzijdse invoer van objecten tussen modulen.

  Wederzijdse invoer van objecten tussen de modulen A en B betekent dat module A objecten invoert van B en omgekeerd. Wederzijdse invoer tussen definitiemodulen is onmogelijk maar is wel toegelaten voor de implementatiemodulen. De wederzijdse invoer wijst op een te sterke koppeling tussen de modulen. Een goed module-ontwerp impliceert een zo los mogelijke koppeling.

- Vermijd de uitvoer van variabelen.

  De uitvoer van variabelen moet worden beschouwd als een uitzondering. Als een gebruikersmodule een variabele invoert, mag deze module slechts naar deze variabele verwijzen voor het raadplegen van de waarde (read only). We kunnen de uitvoer van een variabele vermijden indien we hiervoor in de bibliotheekmodule de gepaste bewerkingen definiëren.

- Vermijd de uitvoer van complexe objecten.

  Als in een module complexe of talrijke objecten worden behandeld, splitsen we deze module bij voorkeur in verschillende modulen, elk met een eigen abstractieniveau.

Voorbeeld :
We definieerden het abstracte type Matrix met de module DynMat. Deze module bevat elementaire bewerkingen op het niveau van de elementen.

```
DEFINITION MODULE DynMat;
EXPORT QUALIFIED
  (* type *) Matrix, DimensieIndex,
  (* proc *) CreeerMatrix, WisMatrix, Hoog, Laag,
             WaardeElement, SchrijfElement;
TYPE DimensieIndex = [1..2];
     Matrix;
...
END DynMat.
```

Op een hoger niveau hebben we de module MatIO gedefinieerd. Deze module specificeert de bewerkingen voor de in- en uitvoer van matrices met type Matrix.

```
DEFINITION MODULE MatIO;
FROM DynMat IMPORT Matrix;
EXPORT QUALIFIED
  (* type *) InProcedure, UitProcedure,
  (* proc *) LeesTabel, DrukTabel;

TYPE InProcedure  = PROCEDURE(VAR CARDINAL, VAR BOOLEAN);
     UitProcedure = PROCEDURE(CARDINAL, VAR BOOLEAN);
...

END MatIO.
```

Een ander voorbeeld is de structuur van de bibliotheekmodulen voor het beheer van bestanden :

Files : algemene bewerkingen op bestanden;

Text : algemene bewerkingen op tekstbestanden;

StandardIO : besturing van de standaardin- en -uitvoer;

SimpleIO : bewerkingen voor de standaardin- en -uitvoer.

Met deze structuur kan een gebruikersmodule een beperkt aantal bewerkingen invoeren en kan ook het abstractieniveau worden gekozen dat past bij de uit te voeren bewerkingen.

- Gebruik proceduretypen

Als we een module ontwerpen voor bewerkingen op een verzameling objecten kunnen we hiervoor zeer algemene bewerkingen definiëren. Specifieke bewerkingen kunnen door de gebruikersmodule worden ingevoerd met behulp van proceduretypen en procedureparameters. In de module MatIO worden de specifieke bewerkingen voor de in- en uitvoer van de elementen ingevoerd met de procedureparameters in en uit. Als een module systeemgekozen bewerkingen bevat, bijvoorbeeld een procedure voor de foutafhandeling, dan voeren we deze procedure uit zodat de gebruikersmodule hiervoor eventueel een eigen verwerking kan definiëren.

Voorbeeld :

```
DEFINITION MODULE A;
EXPORT QUALIFIED
  (* type *) FoutProc,
  (* proc *) VerwerkFout, InstalleerFoutProc, B;
TYPE FoutProc = PROCEDURE(CARDINAL, ARRAY OF CHAR);

(* systeemgekozen procedure voor de foutafhandeling *)
PROCEDURE VerwerkFout(volgnummer : CARDINAL;
                      boodschap  : ARRAY OF CHAR);

PROCEDURE B(x, y : CARDINAL);

PROCEDURE InstalleerFoutProc(foutProc : FoutProc);

END A.


IMPLEMENTATION MODULE A;
(* lokale declaraties *)
VAR fout : FoutProc;

(* implementatie van de procedures *)
PROCEDURE VerwerkFout(volgnummer : CARDINAL;
                      boodschap  : ARRAY OF CHAR);
BEGIN
(* verwerking van de fout *)
...
END VerwerkFout;


PROCEDURE B(x, y : CARDINAL);
VAR succes : BOOLEAN;

BEGIN
...
IF NOT succes
THEN
  fout(x, "foutmelding")
END;
...
END B;


PROCEDURE InstalleerFoutProc(foutProc : FoutProc);
BEGIN
fout := foutProc
END InstalleerFoutProc;
```

```
(* initiëring *)
BEGIN
fout := VerwerkFout
END A.
```

In de implementatiemodule A is een procedurevariabele fout gedefinieerd. Met de initiëring van de module wordt aan deze variabele de waarde VerwerkFout toegekend. Deze waarde is de systeemgekozen waarde. We gebruiken nu de objecten van module A in module M :

```
MODULE M;
FROM A   IMPORT FoutProc, VerwerkFout, InstalleerFoutProc;


PROCEDURE VerwerkLokaleFout(volgnummer : CARDINAL;
                            boodschap  : ARRAY OF CHAR);
BEGIN
(* opdrachten voor de lokale foutafhandeling *)
...
END VerwerkLokaleFout;

(* verwerking M *)
BEGIN
...
(* hier geldt de systeemgekozen foutafhandeling *)
...
InstalleerFoutProc(VerwerkLokaleFout);
(* vanaf hier geldt de lokale foutafhandeling *)
...
InstalleerFoutProc(VerwerkFout);
(* herstel de systeemgekozen foutafhandeling *)
...
END M.
```

Voor het eerste deel van de verwerking van de module M wordt de systeemgekozen procedure voor de foutafhandeling gebruikt bij de aanroep van de procedures van module A. Daarna installeren we de procedure VerwerkLokaleFout. Vanaf dit ogenblik vindt de foutafhandeling in module A met de procedure VerwerkLokaleFout plaats. Tot slot herstellen we de systeemgekozen foutafhandeling.

9.9 Het gebruik van bibliotheekmodulen

De bibliotheekmodulen vormen een essentieel onderdeel van elk in Modula-2 geschreven systeem. De toepassingen van modulen kunnen we in vier categorieën verdelen :

- de module bevat declaraties die van een naam zijn voorzien;
- de module bevat een verzameling samenhangende procedures;
- de module bevat een verzameling gegevens. Deze gegevens zijn slechts toegankelijk door de aanroep van de uitgevoerde procedures;
- de module definieert abstracte gegevenstypen.

We zullen deze categorieën nader bekijken.

Declaraties met een naam

In deze modulen groeperen we logisch samenhangende objecten die globaal zijn voor het programmatuursysteem. Door de declaraties van deze globale objecten in afzonderlijke modulen wordt de onderhoudbaarheid van het systeem bevorderd. Als de definitie moet worden aangepast is de wijziging beperkt tot één enkele module. De wijziging van een globale module impliceert wel dat alle afhankelijke gebruikersmodulen opnieuw moeten worden gecompileerd.

Voorbeeld :

```
DEFINITION MODULE ASCII;
(* Definitie van symbolische constanten voor de ASCII-
   besturingstekens *)
EXPORT QUALIFIED
  (* const *) nul, soh, stx, etx, eot, enq, ack, bel,
              bs,  ht,  lf,  vt,  ff,  cr,  so,  si,
              dle, dcl, dc2, dc3, dc4, nak, syn, etb,
              can, em,  sub, esc, fs,  gs,  rs,  us,
              del;

CONST
  nul = 00C;  soh = 01C;  stx = 02C;  etx = 03C;
  eot = 04C;  enq = 05C;  ack = 06C;  bel = 07C;
  bs  = 10C;  ht  = 11C;  lf  = 12C;  vt  = 13C;
  ff  = 14C;  cr  = 15C;  so  = 16C;  si  = 17C;
  dle = 20C;  dcl = 21C;  dc2 = 22C;  dc3 = 23C;
  dc4 = 24C;  nak = 25C;  syn = 26C;  etb = 27C;
  can = 30C;  em  = 31C;  sub = 32C;  esc = 33C;
  fs  = 34C;  gs  = 35C;  rs  = 36C;  us  = 37C;
  del = 177C;
END ASCII.
```

Verzameling samenhangende procedures

Deze modulen behandelen over het algemeen twee verschillende representaties van gegevens en bevatten de procedures voor het omzetten van de ene naar de andere representatie. In deze modulen worden geen eigen gegevensstructuren gedefinieerd. Enkele voorbeelden van dit type module zijn : een module met wiskundige functies, een module voor de conversie van de interne representatie van gegevens naar tekst of omgekeerd (standaardmodulen Convert, ConvertReal, MathLib).

Voorbeeld :

```
DEFINITION MODULE ConvertReal;
(* conversie van variabelen met type REAL naar een string en
   omgekeerd
*)

EXPORT QUALIFIED
  (* proc *) RealToStr, StrToReal;

PROCEDURE RealToStr(     real      : REAL;
                     VAR str       : ARRAY OF CHAR;
                         breedte   : CARDINAL;
                         decPlaats : INTEGER;
                     VAR succes    : BOOLEAN);

PROCEDURE StrToReal(     str       : ARRAY OF CHAR;
                     VAR real      : REAL;
                     VAR succes    : BOOLEAN);

END ConvertReal.
```

Module met ingebouwde gegevensstructuren

De essentie van de module is een aantal ingebouwde gegevensstructuren. De details van de gegevensstructuren worden verborgen voor de gebruikersmodulen. De gegevensstructuren zijn slechts toegankelijk via de bewerkingen die door de module worden uitgevoerd. Enkele voorbeelden van dit type module zijn de bibliotheekm odulen Storage voor het beheer van het geheugen en Terminal voor de aansturing van het beeldscherm. Ook de module FreqTab behoort tot dit moduletype.

Voorbeeld :

```
DEFINITION MODULE FreqTab;
EXPORT QUALIFIED
  (* proc *) TelFrequentie, Frequentie;

PROCEDURE TelFrequentie(teken : CHAR);
PROCEDURE Frequentie(teken : CHAR) : CARDINAL;

END FreqTab.
```

De frequentietabel wordt gedefinieerd en geïnitieerd in de implementatiemodule. Alle bewerkingen op de tabel worden uitgevoerd via de procedures TelFrequentie en Frequentie.

De module definieert abstracte gegevenstypen

De uitvoer van de module is een gegevenstype en de bewerkingen die voor dit type zijn gedefinieerd. De details van het gegevenstype en van de bewerkingen worden verborgen voor de gebruikersmodulen. Met een verborgen gegevenstype kunnen slechts bewerkingen worden uitgevoerd binnen de implementatiemodule waardoor de betrouwbaarheid beter kan worden verzekerd. Bij het vorige moduletype bevat de implementatiemodule de variabelen met de verborgen gegevensstructuur, bij dit moduletype worden de variabelen met een verborgen type in de gebruikersmodulen zelf gedeclareerd. Een voorbeeld van dit type is de module Files waarin het abstracte gegevenstype File wordt gedefinieerd. We definieerden ook reeds het abstracte gegevenstype Matrix met basistype CARDINAL.

Literatuur

Booch G., 'Systeemontwikkeling met Ada',
oorspronkelijke titel : Software Engineering with Ada,
Academic Service 1985

Bron C., Dijkstra E.W., 'A Note on the Checking of Interfaces Between Separately Compiled Modules',
Sigplan, augustus 1985

Coar D., 'Pascal, Ada and Modula-2'
BYTE, augustus 1984

Cornelius B., 'The Scope Problem Caused by Modules',
Modula-2 News, juli 1985

Gutknecht J., 'Tutorial on Modula-2',
BYTE, augustus 1984

Marini G., McLarty H., 'Modula-2 Aids Structured Programming',
Mini-Micro Systems, mei 1985

McCormack J., Gleaves R., 'Modula-2, A Worthy Successor to Pascal',
BYTE, augustus 1982

Spector D., 'Ambiguities and Insecurities in Modula-2',
Sigplan, augustus 1982

Wirth N., 'Revisions and Amendements to Modula-2',
ETH Zurich, 1984

Wirth N., 'History and Goals of Modula-2',
BYTE, augustus 1984

Oefeningen

1. Beschouw de volgende definitie- en implementatiemodulen :

```
DEFINITION MODULE Getallen;
EXPORT QUALIFIED
   (* type *) Hoeveelheid;
TYPE Hoeveelheid;
END Getallen.

IMPLEMENTATION MODULE Getallen;
TYPE Hoeveelheid = CARDINAL;
END Getallen.
```

en de programmamodule :

```
MODULE Produkt;
FROM Getallen IMPORT Hoeveelheid;
VAR a : Hoeveelheid;
BEGIN
a := a * 3
END Produkt.
```

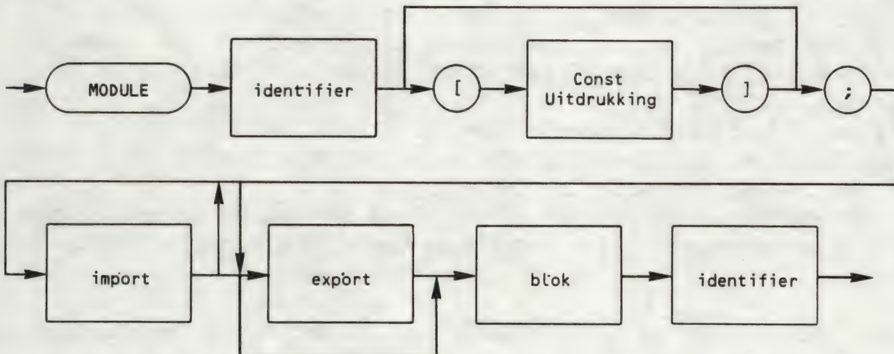
Is de toekenningsopdracht in Produkt uitvoerbaar ? Onderzoek waarom of waarom niet. Pas zo nodig de modulen aan.

2. Ontwerp en implementeer een module FreqTab waarin zowel grote als kleine letters worden geteld.

10 Interne modulen

Een interne module wordt genest in een procedure of in een andere module. De interne module behoort steeds tot een programma- of een implementatiemodule. Met een interne module creëren we een geïsoleerde omgeving waardoor het bereik en de toegankelijkheid van de identifiers kan worden beheerst en beheerd. De syntaxis voor de interne module is :

```
ModuleDeclaratie = 'MODULE' identifier ';'
                   { import }
                   [ export ]
                   blok
                   identifier
```



10.1 De in- en uitvoer van objecten

Elke module bepaalt het bereik en de toegankelijkheid van de identifiers die in de module worden gebruikt. Ook een procedure definieert het bereik van de lokale declaraties. De levensduur van de lokale variabelen in een procedure is echter beperkt tot de

duur van de activiteit van de procedure, terwijl de levensduur van variabelen in een module gelijk is aan de levensduur van het omhullende blok. Zo is bijvoorbeeld de levensduur van een variabele, die gedeclareerd is in een interne module van de programmamodule, gelijk aan de levensduur van het programma. Ook kan het bereik van de objecten, die in een module worden gedeclareerd, buiten de module worden uitgebreid met een in- en/of uitvoerlijst. De syntaxis voor deze in- en uitvoerlijsten is :

```
import = [ 'FROM' identifier ] 'IMPORT' IdentLijst ';'

export = 'EXPORT' [ 'QUALIFIED '] IdentLijst ';'
```

10.2 Het bereik van identifiers

We geven eerst de regels die gelden voor het bereik en de in- en uitvoer van identifiers :

- een identifier, gedeclareerd in een module M, behoort ook tot het bereik van de lokale procedures in M;
- het bereik van een identifier, behorende tot het bereik van een module, behoort niet tot het bereik van de interne modulen;
- het bereik van een identifier, behorende tot het bereik van een module M, wordt uitgebreid tot de omhullende module indien de identifier in de uitvoerlijst van M wordt vermeld. Evenzo wordt het bereik van een identifier uitgebreid tot een interne module N indien de identifier in de invoerlijst van N wordt vermeld;
- naar een identifier in een uitvoerlijst met een QUALIFIED-clausule wordt buiten de module verwezen door gebruik te maken van een gekwalificeerde identifier;
- met de FROM-clausule in een invoerlijst moeten we de modulenaam niet meer vermelden bij het gebruik van de ingevoerde identifiers.

We illustreren nu elk van deze regels met enkele voorbeelden. Beschouw de eerste twee regels :

- een identifier, gedeclareerd in een module M, behoort ook tot het bereik van de lokale procedures in M;
- het bereik van een identifier, behorend tot het bereik van de module, behoort niet tot het bereik van de interne modulen.

In een module M kunnen we globale objecten declareren. De identifiers van deze objecten zijn ook toegankelijk in de procedures die in M worden gedeclareerd, maar niet in de interne modulen die in M worden gedeclareerd.

Voorbeeld :
Beschouw module A waarin een interne module B en een procedure C worden gedeclareerd. In de programmatekst zijn regelnummers toegevoegd om de verwijzingen aan te geven.

```
 1 MODULE A;
 2 VAR  a1 : CHAR;
 3
 4 MODULE B;
 5   VAR b1, b2 : CHAR;
 6   BEGIN
 7   ...
 8 END B;
 9
10 PROCEDURE C;
11   VAR c1 : CHAR;
12   BEGIN
13   ...
14 END C;
15 BEGIN
16 ...
17 END A.
```

De variabele a1 is gedeclareerd in module A. Het bereik van a1 is regels 2 en 3 en de regels 9 tot 17 (regel 17 niet inbegrepen). De variabelen b1 en b2 zijn gedeclareerd in module B. Het bereik omvat regel 5 tot regel 8. De variabele c1 is gedeclareerd in procedure C en heeft een bereik van regel 11 tot regel 14. De levensduur van de variabele a1 is de duur van module A. Module A omsluit module B. De levensduur van de variabelen b1 en b2 is derhalve de duur van module A. De levensduur van c1 is de tijd dat procedure C actief is.
Met de module B definiëren we een beperkt bereik voor de variabelen b1 en b2. De levensduur van deze variabelen is echter de levensduur van module A.

Beschouw nu de regel :
- het bereik van een identifier, behorende tot het bereik van een module, wordt uitgebreid tot de omhullende module indien de identifier in de uitvoerlijst van M wordt vermeld.

We bekijken weer de module A met de interne module B. De module B bevat echter nu een uitvoerlijst met de identifier b1.

```
 1 MODULE A;
 2 VAR  a1 : CHAR;
 3
```

```
 4 MODULE B;
 5   EXPORT b1;
 6   VAR b1, b2 : CHAR;
 7   BEGIN
 8   ...
 9 END B;
10
11 PROCEDURE C;
12   VAR c1 : CHAR;
13   BEGIN
14   ...
15 END C;
16 BEGIN
17 ...
18 END A.
```

Het bereik van de identifier b1 wordt nu uitgebreid tot de omhullende module A. Het nieuwe bereik voor b1 is nu regel 2 tot regel 18. Het bereik van b2 is blijft ongewijzigd.

We passen deze eigenschappen toe in het volgende voorbeeld : de programmamodule A drukt honderd willekeurige getallen af.

```
MODULE A;
FROM SimpleIO IMPORT WriteCard, WriteLn;

  MODULE WillekeurigGetal;
  EXPORT Willekeurig;
  VAR start : CARDINAL;

  PROCEDURE Willekeurig () : CARDINAL;
  CONST modulus = 7415;
        stap    = 25543;
  BEGIN
  start := (start + stap) MOD modulus;
  RETURN start
  END Willekeurig;

  BEGIN
  start := 31415
  END WillekeurigGetal;

VAR i : [1..100];
BEGIN
FOR i := 1 TO 100 DO
  WriteCard(Willekeurig(),5);
  WriteLn
END
END A.
```

De functieprocedure Willekeurig levert een willekeurig getal af. Elke nieuwe waarde wordt berekend aan de hand van de vorige waarde met de opdracht

```
start := (start + stap) MOD modulus
```

De waarde van de variabele start moeten we behouden tussen de opeenvolgende aanroepen van de procedure Willekeurig. Door de module WillekeurigGetal is de levensduur van de variabele start gelijk aan de duur van het programma (module A). Het bereik van start wordt echter beperkt tot de initiëring en de procedure Willekeurig. Het bereik van de identifier Willekeurig moet echter worden uitgebreid tot de omhullende module. We vermelden daarom Willekeurig in de uitvoerlijst van de module WillekeurigGetal.

Deze oplossing heeft een belangrijk nadeel : bij elke uitvoering van de programmamodule A worden telkens dezelfde getallen afgedrukt. Als we echter aan de variabele start een willekeurige waarde kunnen toekennen bij het begin van het programma, worden bij elke uitvoering steeds honderd andere getallen afgedrukt. We formuleren hiervoor zo dadelijk een mogelijke oplossing. We beschouwen eerst de volgende regel :

- het bereik van een identifier wordt uitgebreid tot een interne module N indien de identifier in de invoerlijst van N wordt vermeld.

We geven nog eens de module A met de interne module B. De module B bevat echter nu een invoerlijst met de identifier a1.

```
MODULE A;
VAR  a1 : CHAR;

MODULE B;
  IMPORT a1;
  VAR b1, b2 : CHAR;
  BEGIN
  ...
END B;

PROCEDURE C;
  VAR c1 : CHAR;
  BEGIN
  ...
END C;
BEGIN
...
END A.
```

Het bereik van de variabele al wordt zo uitgebreid tot het bereik van de interne module B. Het nieuwe bereik van de variabele al is nu regel 2 tot 18.

We passen deze eigenschap toe op het programmavoorbeeld. We veronderstellen in de programmamodule A een variabele SysteemTijd met type CARDINAL. De waarde van deze variabele is een voorstelling van de tijd als een natuurlijk getal. De waarde is dus afhankelijk van het ogenblik waarop het programma wordt gestart. Als we deze waarde gebruiken als de startwaarde voor de variabele start, worden bij elke activering van het programma andere toevalsgetallen afgedrukt. De variabele SysteemTijd is bereikbaar in de module WillekeurigGetal als we de identifier vermelden in een invoerlijst. Dit geeft de volgende oplossing :

```
MODULE A;
FROM SimpleIO IMPORT WriteCard, WriteLn;
VAR SysteemTijd : CARDINAL;

  MODULE WillekeurigGetal;
  IMPORT SysteemTijd;
  EXPORT Willekeurig;
  VAR start : CARDINAL;

  PROCEDURE Willekeurig () : CARDINAL;
  CONST modulus = 7415;
        stap    = 25543;
  BEGIN
  start := (start + stap) MOD modulus;

  RETURN start
  END Willekeurig;

  BEGIN
  start := SysteemTijd
  END WillekeurigGetal;

VAR i : [1..100];
BEGIN
(* initiëring SysteemTijd *)
...
(* afdrukken van de getallen *)
FOR i := 1 TO 100 DO
  WriteCard(Willekeurig(),5);
  WriteLn
END
END A.
```

Modulen kunnen tot een willekeurige diepte worden genest. Met de in- en uitvoerlijst wordt het bereik van een identifier telkens uitgebreid tot het bereik van de interne of van de omhullende module.

Voorbeeld :

```
 1 MODULE A;
 2   MODULE B;
 3   EXPORT k;
 4     MODULE C;
 5     EXPORT k;
 6     VAR k : CHAR;
 7     ...
 8     END C;
 9   END B;
10 END A.
```

De variabele k wordt gedefinieerd in de module C. Deze module is genest in module B en module B is genest in module A. Het bereik van k wordt uitgebreid tot de omhullende module B door de uitvoerlijst in regel 5. Het bereik wordt opnieuw uitgebreid van module B tot de omhullende module A doordat k ook in de uitvoerlijst van module B voorkomt (in regel 3).

Voorbeeld :

```
 1 MODULE A;
 2 VAR k : CHAR;
 3   MODULE B;
 4   IMPORT k;
 5     MODULE C;
 6     IMPORT k;
 7     ...
 8     END C;
 9   END B;
10 END A.
```

De variabele k wordt gedeclareerd in de module A. Het bereik van k wordt uitgebreid tot de interne module B door de invoerlijst in regel 4. Daarna wordt het bereik uitgebreid tot module C door de invoerlijst in regel 6.

We passen deze eigenschap nu toe op het programma voor het afdrukken van willekeurige getallen. In de meeste computersystemen wordt de systeemtijd in een systeemafhankelijke bibliotheekmodule gedefinieerd. Bijvoorbeeld : de systeemtijd wordt gedefinieerd in een module System. Eerst voeren we de identifier SysteemTijd in de programmamodule A in met

```
FROM System IMPORT SysteemTijd;
```

De identifier SysteemTijd behoort nu tot het bereik van module A. Dit bereik wordt uitgebreid tot de module WillekeurigGetal door

```
IMPORT SysteemTijd;
```

Dit geeft de volgende oplossing :

```
MODULE A;
FROM SimpleIO IMPORT WriteCard, WriteLn;
FROM System   IMPORT SysteemTijd;
  MODULE WillekeurigGetal;
  IMPORT SysteemTijd;
  EXPORT Willekeurig;
  VAR start : CARDINAL;

  PROCEDURE Willekeurig () : CARDINAL;
  CONST modulus = 7415;
        stap    = 25543;
  BEGIN
  start := (start + stap) MOD modulus;
  RETURN start
  END Willekeurig;

  BEGIN
  start := SysteemTijd
  END WillekeurigGetal;

VAR i : [1..100];
BEGIN
(* afdrukken van de getallen *)
FOR i := 1 TO 100 DO
  WriteCard(Willekeurig(),5);
  WriteLn
END
END A.
```

Beschouw nu de volgende eigenschappen :
- naar een identifier in een uitvoerlijst met een QUALIFIED-clausule kan buiten de module worden verwezen door gebruik te maken van een gekwalificeerde identifier. Een gekwalificeerde identifier is samengesteld uit een modulenaam, een punt '.' en een identifier. De modulenaam is de naam van de module met een bereik waartoe de identifier behoort;
- met de FROM-clausule in een invoerlijst moeten we modulenaam niet meer vermelden bij het gebruik van de ingevoerde identifiers.

Voorbeeld :

```
MODULE P;
...
  MODULE M1;
  EXPORT QUALIFIED a, b, c;
  VAR a, b, c : CHAR;
  ...
  END M1;

  MODULE M2;
  EXPORT QUALIFIED a, b, c;
  VAR a, b, c : CARDINAL;
  ...
  END M2;
...
BEGIN
M1.a := 'a';
M2.a := 10;
...
END P.
```

Zowel in de module M1 als de module M2 bevatten de uitvoerlijsten de identifiers a, b en c. Het bereik van deze identifiers wordt dus uitgebreid tot de omhullende module P. Met de QUALIFIED-clausule verwijzen we in de omhullende module naar de identifiers van module M1 :

M1.a M1.b M1.c

En naar de identifiers van module M2 met

M2.a M2.b M2.c

Zonder deze clausule kunnen de identifiers a, b en c van module M1 of M2 in module P niet van elkaar worden onderscheiden. We gebruiken de QUALIFIED-clausule om dergelijke conflicten in de omhullende module te voorkomen. Met de FROM-clausule in een invoerlijst worden de identifiers in de module gebruikt alsof ze niet in een QUALIFIED-clausule in de uitvoerlijst voorkomen.

Voorbeeld :

```
MODULE P;
...
  MODULE M1;
  EXPORT QUALIFIED a, b, c;
  VAR a, b, c : CHAR;
  ...
  END M1;
```

```
  MODULE M2;
  EXPORT a, b, c;
  VAR a, b, c : CARDINAL;
  ...
  END M2;

  MODULE M3;
  FROM M1 IMPORT a;
  BEGIN
  a := 'a'
  END M3;
...
BEGIN
M1.c := 'c';
a := 10;
...
END P.
```

In de module M1 worden de identifiers a, b en c uitgevoerd met een QUALIFIED-clausule. In de omhullende module verwijzen we naar deze identifiers met een gekwalificeerde identifier, bijvoorbeeld

M1.c := 'c'

De uitvoerlijst van de module M2 bevat de identifiers a, b en c. Het bereik van deze identifiers wordt dus uitgebreid tot de omhullende module P. We verwijzen in module P naar deze identifiers gewoon met a, b of c. In module M3 verwijzen we naar module M1. Deze module moet behoren tot het bereik van de module die M3 omhult. De identifier a moet voorkomen in de uitvoerlijst van module M1. Door de invoerlijst met een FROM-clausule kunnen we toch naar identifier a verwijzen zonder de kwalificatie van de modulenaam M1, bijvoorbeeld :

a := 'a'

10.3 Impliciete in- en uitvoer

We beschreven reeds de impliciete in- en uitvoer voor een recordtype en een enumeratietype in het hoofdstuk 'Bibliotheekmodulen'. De regels gelden ook voor de interne modulen :

- als een recordtype wordt in- of uitgevoerd, worden ook de identifiers van de recordvelden in- of uitgevoerd. Deze regel geldt ook voor de constante identifiers van een enumeratietype.

Voor de interne modulen kunnen we hieraan nog de volgende regel toevoegen :

- als een modulenaam M wordt in- of uitgevoerd, behoort deze naam tot het bereik van het interne of het omhullende blok. In dit nieuwe bereik kunnen we naar de identifiers, die door M worden uitgevoerd, verwijzen door gebruik te maken van een gekwalificeerde identifier.

Voorbeeld :

```
MODULE M;
  MODULE A;
  EXPORT B;
    MODULE B;
    EXPORT i, j;
    VAR i, j : CARDINAL;
    END B;
  BEGIN
  i := 1;
  j := 10;
  END A;
BEGIN
B.i := 20;
B.j := 0
END M.
```

De module A is genest in module M, de module B is genest in module A. In module B worden de identifiers i en j gedeclareerd. Het bereik van deze identifiers wordt met de uitvoerlijst

EXPORT i, j;

uitgebreid tot het bereik van de omhullende module A. In module A verwijzen we naar deze identifiers met i en j. De module A bevat opnieuw een uitvoerlijst waarin de modulenaam B wordt vermeld. Het bereik van de modulenaam B wordt hierdoor uitgebreid tot de omhullende module M. Het bereik van de identifiers in de uitvoerlijst van module B wordt impliciet uitgebreid tot module M. In deze module verwijzen we hiernaar met de gekwalificeerde identifiers

B.i en B.j

Beschouw een soortgelijk voorbeeld waarbij een modulenaam wordt ingevoerd :

```
MODULE M;
  MODULE A;
  EXPORT a, b, c;
  VAR a, b, c : CARDINAL;
  ...
  END A;

  MODULE B;
  IMPORT A;
  BEGIN
  A.a := A.b + A.c
  END B;
...
END M.
```

De module A bevat in de uitvoerlijst de identifiers a, b en c. Het bereik van deze identifiers wordt hierdoor uitgebreid tot de omhullende module M maar niet tot de interne module B. In module B wordt de modulenaam A ingevoerd. De identifiers in de uitvoerlijst van A worden impliciet ingevoerd in module B. In deze module verwijzen we hiernaar met

A.a A.b en A.c

Toepassing : de afdruk van een dynamische matrix

We illustreren het gebruik van een interne module in de volgende toepassing : de elementen van een matrix worden gelezen van de standaardinvoer, een element per regel. De elementen worden afgedrukt op de standaarduitvoer in de vorm van een tabel. Dit betekent dat elke rij van de matrix op een regel wordt afgedrukt. We gebruiken de bibliotheekmodulen DynMat en MatIO. In deze laatste module zijn de typen InProcedure en UitProcedure en de bewerkingen LeesTabel en DrukTabel gedefinieerd als :

```
TYPE InProcedure  = PROCEDURE(VAR element : CARDINAL,
                              VAR succes  : BOOLEAN);
     UitProcedure = PROCEDURE(    element : CARDINAL;
                              VAR succes  : BOOLEAN);

PROCEDURE LeesTabel(    m      : Matrix;
                    VAR succes : BOOLEAN;
                        in     : InProcedure);
```

```
PROCEDURE DrukTabel(    m       : Matrix;
                    VAR succes : BOOLEAN;
                        uit    : UitProcedure);
```

We definiëren nu de procedures LeesElement met type InProcedure en de procedure SchrijfElement met type UitProcedure.

```
PROCEDURE LeesElement(VAR element : CARDINAL;
                      VAR succes  : BOOLEAN);
```

De invoer voor de procedure LeesElement is een getal per regel. De procedures SimpleIO.ReadCard en SimpleIO.ReadLn kunnen worden gebruikt voor de invoer van een element. De declaratie van LeesElement is dus :

```
PROCEDURE LeesElement(VAR element : CARDINAL;
                      VAR succes  : BOOLEAN);
BEGIN
SimpleIO.ReadCard(element, succes);
SimpleIO.ReadLn
END LeesElement;
```

```
PROCEDURE SchrijfElement(    element : CARDINAL;
                         VAR succes  : BOOLEAN);
```

De procedure SchrijfElement schrijft de elementen van een rij van een matrix op een regel. Als een rij is afgedrukt, wordt met een nieuwe regel begonnen voor de volgende rij. De procedure Schrijf-Element moet dus onthouden wanneer het laatste element van een rij is afgedrukt en moet hiervoor het aantal elementen per rij kennen. De variabele elementTeller telt het aantal elementen dat op een regel is afgedrukt, de variabele aantalKolommen is gelijk aan het aantal elementen per regel. Deze beide variabelen moeten hun waarden behouden tussen de opeenvolgende aanroepen van de procedure SchrijfElement. We declareren daarom elementTeller en aantalKolommen als globale variabelen.

```
VAR aantalKolommen : CARDINAL;
    elementTeller  : CARDINAL;

PROCEDURE SchrijfElement(    element : CARDINAL;
                         VAR succes  : BOOLEAN);
BEGIN
SimpleIO.WriteCard(element,5);
succes := TRUE;
```

```
IF elementTeller = aantalKolommen
THEN
  elementTeller := 1;
  SimpleIO.WriteLn
ELSE
  INC(elementTeller)
END
END SchrijfElement;
```

Voordat de procedure DrukTabel de procedure SchrijfElement aanroept, moeten de variabelen aantalKolommen en elementTeller worden geïnitieerd. We declareren hiervoor de procedure InitieerAfdruk :

```
PROCEDURE InitieerAfdruk(aantal : CARDINAL);
BEGIN
aantalKolommen := aantal;
elementTeller  := 1
END InitieerAfdruk;
```

Het bereik van de globale variabelen aantalKolommen en elementTeller kan worden beperkt tot de procedures SchrijfElement en InitieerAfdruk. Deze variabelen en procedures worden ingekapseld in een interne module StandaardMatIO. De procedure SchrijfElement verwijst naar de bibliotheekmodule SimpleIO. Deze module wordt derhalve ingevoerd met een invoerlijst

```
IMPORT SimpleIO
```

De bewerkingen SchrijfElement en InitieerAfdruk worden gebruikt buiten de module StandaardMatIO. We vermelden deze identifiers in een uitvoerlijst

```
EXPORT SchrijfElement, InitieerAfdruk;
```

De volledige declaratie van de interne module StandaardMatIO is dan :

```
MODULE StandaardMatIO;
IMPORT SimpleIO;
EXPORT SchrijfElement, InitieerAfdruk;

VAR aantalKolommen : CARDINAL;
    elementTeller  : CARDINAL;
```

```
PROCEDURE SchrijfElement(    element : CARDINAL;
                         VAR succes  : BOOLEAN);
BEGIN
SimpleIO.WriteCard(element, 5);
succes := TRUE;
IF elementTeller = aantalKolommen
THEN
  elementTeller := 1;
  SimpleIO.WriteLn
ELSE
  INC(elementTeller)
END
END SchrijfElement;

PROCEDURE InitieerAfdruk(aantal : CARDINAL);
BEGIN
aantalKolommen := aantal;
elementTeller  := 1
END InitieerAfdruk;

END StandaardMatIO;
```

We beschikken nu over de procedures LeesElement en SchrijfElement voor de in- en de uitvoer van een matrix volgens de gegeven specificaties. We formuleren nu de volledige programmamodule waarbij de elementen van een 3 * 4 matrix elk afzonderlijk regel voor regel worden ingevoerd en waarbij de matrix als een tabel wordt afgedrukt.

```
MODULE MatTab;
FROM DynMat  IMPORT Matrix, Laag, Hoog, CreeerMatrix;
FROM MatIO   IMPORT DrukTabel, LeesTabel;
IMPORT SimpleIO;

  MODULE StandaardMatIO;
  IMPORT SimpleIO;
  EXPORT SchrijfElement, InitieerAfdruk;

  VAR aantalKolommen : CARDINAL;
      elementTeller  : CARDINAL;

  PROCEDURE SchrijfElement(    element : CARDINAL;
                           VAR succes  : BOOLEAN);
```

```
  BEGIN
  SimpleIO.WriteCard(element, 5);
  succes := TRUE;
  IF elementTeller = aantalKolommen
  THEN
    elementTeller := 1;
    SimpleIO.WriteLn
  ELSE
    INC(elementTeller)
  END
  END SchrijfElement;


  PROCEDURE InitieerAfdruk(aantal : CARDINAL);
  BEGIN
  aantalKolommen := aantal;
  elementTeller  := 1
  END InitieerAfdruk;

END StandaardMatIO;


PROCEDURE LeesElement(VAR element : CARDINAL;
                      VAR succes  : BOOLEAN);
BEGIN
SimpleIO.ReadCard(element, succes);
SimpleIO.ReadLn
END LeesElement;


(* declaraties programmamodule *)
VAR m  : Matrix;
    ok : BOOLEAN;

(* verwerking programmamodule *)
BEGIN
CreeerMatrix(1, 3, 1, 4, m);
LeesTabel(m, ok, LeesElement);
InitieerAfdruk(Hoog(m, 2) - Laag(m, 2) + 1);
DrukTabel(m, ok, SchrijfElement)
END MatTab.
```

Literatuur

Cornelius B., 'Problems with the Report on Modula-2', The Modus Quarterly, juli 1985

Cornelius B., 'The Scope Problem Caused by Modules', The Modus Quarterly, juli 1985

Wirth N., 'Programming in Modula-2', derde verbeterde druk, Springer-Verlag, Berlijn 1985

Oefeningen

1. Bepaal de uitvoer van de volgende programmamodule :

```
MODULE A;
IMPORT SimpleIO;
  MODULE A1;
  IMPORT SimpleIO;
  EXPORT
    (* var *) a, b, c;
    MODULE A2;
    IMPORT SimpleIO;
    EXPORT
      (* var *) d, e, f;
    VAR d, e, f  : CHAR;

    BEGIN
    d := 'f';
    e := 'u';
    f := 'n';
    SimpleIO.WriteLn;
    SimpleIO.WriteChar(d);
    SimpleIO.WriteChar(e);
    SimpleIO.WriteChar(f)
    END A2;
  VAR a, b, c : INTEGER;
  BEGIN
  a := 1;
  b := 2;
  c := 3;
  SimpleIO.WriteLn;
  SimpleIO.WriteInt(a, 3);
  SimpleIO.WriteInt(b, 3);
  SimpleIO.WriteInt(c, 3);
  SimpleIO.WriteLn
  END A1;
```

```
BEGIN
SimpleIO.WriteLn;
SimpleIO.WriteString('De uitvoer van dit programma is : ')
END A.
```

2. Bepaal de uitvoer van de volgende programmamodule :

```
MODULE A;
IMPORT SimpleIO;

PROCEDURE B;
  MODULE A1;
  IMPORT SimpleIO;
  EXPORT
    (* var *) a, b, c;
    MODULE A2;
    IMPORT SimpleIO;
    EXPORT
      (* var *) d, e, f;
    VAR d, e, f  : CHAR;
    BEGIN
    d := 'f';
    e := 'u';
    f := 'n';
    SimpleIO.WriteLn;
    SimpleIO.WriteChar(d);
    SimpleIO.WriteChar(e);
    SimpleIO.WriteChar(f)
    END A2;

  VAR a, b, c : INTEGER;
  BEGIN
  a := 1;
  b := 2;
  c := 3;
  SimpleIO.WriteLn;
  SimpleIO.WriteInt(a, 3);
  SimpleIO.WriteInt(b, 3);
  SimpleIO.WriteInt(c, 3);
  SimpleIO.WriteLn
  END A1;
END B;

BEGIN
SimpleIO.WriteLn;
SimpleIO.WriteString('De uitvoer van dit programma is : ')
END A.
```

11 De Modula-2 standaardbibliotheek

De woordenschat van Modula-2 is beperkt, maar kan gemakkelijk worden uitgebreid door het toevoegen van objecten die in bibliotheekmodulen worden gedefinieerd. Bij het ontwikkelen van programma's streven we de overdraagbaarheid van onze programma's na zodat de ontwikkelingskosten kunnen worden beperkt. Het gebruik van een standaardbibliotheek is wellicht even belangrijk als het gebruik van een gestandaardiseerde programmeertaal. Door de standaardbibliotheek wordt beoogd de overdraagbaarheid van programma's tussen verschillende machines met dezelfde omgeving of zelfs met een verschillend besturingssysteem te bevorderen.

De eerste implementaties van Modula-2 in de Technische Hogeschool ETH van Zurich hadden elk een eigen, verschillende bibliotheek. Ook bij de eerste commerciële implementaties bestond er geen eensgezindheid over de bibliotheekmodulen. Medio 1983 is een werkgroep opgericht om de verschillende bibliotheken op elkaar af te stemmen en te komen tot een ontwerp voor een standaardbibliotheek. De activiteiten van deze werkgroep resulteerden in december 1984 in een ontwerp van een bibliotheek. Dit ontwerp is gepubliceerd in het tijdschrift 'MODUS Quarterly' van de Modula-2 gebruikersvereniging. Verschillende systeemhuizen die Modula-2 implementeren, hebben reeds een implementatie met de standaardbibliotheek aangekondigd. In dit boek beschrijven we deze standaardbibliotheek. Ze is ook geïmplementeerd voor een bestaande commerciële Modula-2 implementatie.

Doel van de standaardbibliotheek

De standaardbibliotheek moet ons ertoe in staat stellen programma's te ontwerpen die overdraagbaar zijn tussen verschillende Modula-2 implementaties. De bibliotheek levert faciliteiten voor het programmeren van programmatuurhulpmiddelen (Engels : software tools) en toepassingen op het terrein van de gegevensverwerking (Engels : data processing). Voor dit type toepassingen kunnen we met de bibliotheek algoritmen voor een standaardomgeving beschrijven. De bibliotheek ondersteunt niet het programmeren van grafische toepassingen en communicatieprogrammatuur en ook niet het

ontwikkelen van functies voor het besturingssysteem (takenbeheer).
Voor programmeurs met Pascal-ervaring is de bibliotheek gemakkelijk te begrijpen; de vereiste kennis van de onderliggende omgeving is beperkt en moeilijke constructies, zoals bijvoorbeeld het opvangen van uitvoeringsfouten, behoren niet tot de bibliotheek.

Overzicht van de standaardbibliotheek

De standaardbibliotheek bevat modulen voor het besturen van de in- en uitvoer, voor conversies en wiskundige functies, voor het dynamische beheer van het geheugen en voor de aanroep van programma's. De in- en uitvoer worden gedefinieerd in verschillende modulen. Elk van deze modulen heeft een eigen abstractieniveau en vormt aldus een laag in het in- en uitvoersysteem. Deze modulen zijn afhankelijk van elkaar gedefinieerd. De overige modulen zijn onafhankelijk van elkaar.

We geven een kort alfabetisch overzicht van de modulen en we vermelden voor elke module het verband met de overige modulen :

Binary : Met de module Binary kunnen we binaire bestanden lezen en schrijven. Een binair bestand is niet geformatteerd. De in- en uitvoer wordt volledig bepaald door de toepassing.

Convert : De module Convert levert procedures voor het omzetten van getallen met type CARDINAL of INTEGER van de interne representatie naar een uitwendige, leesbare representatie en omgekeerd.

ConvertReal : Deze module breidt de mogelijkheden van de module Convert uit tot getallen met type REAL.

Directory : Met deze module krijgen we toegang tot de inhoudstabellen van bestanden. De module bevat bewerkingen voor het manipuleren van bestanden op basis van de bestandsnaam.

FilePositions : Door deze module kunnen we een willekeurige plaats in een bestand berekenen en adresseren. Als we de bewerkingen van deze module combineren met die van de module Text of Binary, kunnen we een bestand lezen of schrijven in een niet-sequentiële volgorde.

Files : Files is de basismodule voor het beheer van de bestanden. In deze module wordt het verborgen type File gedefinieerd. De modulen Text, Binary en FilePositions zijn gedefinieerd op basis van deze module.

MathLib : Deze module levert de procedures voor het uitvoeren van wiskundige functies.

NumberIO : Deze module levert de bewerkingen voor de in- en uitvoer van getallen met type INTEGER, CARDINAL en ook van getallen met een grondtal begrepen tussen 2 en 31.

Program : De module Program maakt het ons mogelijk vanuit een actief Modula-2 programma andere Modula-2 programma's aan te roepen.

SimpleIO : De module SimpleIO definieert de bewerkingen voor de in- en uitvoer van tekstbestanden van en naar de standaardin- en -uitvoer. Deze bewerkingen zijn gemakkelijker te gebruiken dan de overeenkomstige bewerkingen van de module Text.

Storage : Deze module bevat de bewerkingen voor het dynamische beheer van het geheugen. Deze bewerkingen zijn noodzakelijk bij het aanroepen van de standaardprocedures NEW en DISPOSE.

String : De module String bevat bewerkingen voor het uitvoeren van bewerkingen met stringvariabelen.

Terminal : Door de module Terminal krijgen we rechtstreeks toegang tot de in- en uitvoer van het werkstation. Bij de modulen Text en SimpleIO vindt de in- en uitvoer via het systeem voor het beheer van bestanden plaats.

Text : Door de module Text kunnen we een geformatteerd tekstbestand lezen en schrijven. Een tekstbestand is dan een opeenvolging van tekens, strings of regels.

Opmerking : In de beschrijving van de bibliotheek worden de formele parameters tussen de haakjes '<' en '>' in de tekst vermeld. Bijvoorbeeld :

```
PROCEDURE Assign(    bron   : ARRAY OF CHAR;
                 VAR doel   : ARRAY OF CHAR;
                 VAR succes : BOOLEAN);
```

De formele parameters voor de procedure Assign zijn <bron>, <doel> en <succes>.

Literatuur

Bondy, J., 'Modula-2 Library Documentation',
The Modus Quarterly 1, januari 1985

Bush, R., 'Modula-2 Library Rationale',
The Modus Quarterly 1, januari 1985

Djavaheri M., 'Standard Library for the Unix OS',
The Modus Quarterly 3, november 1985

Nagler, R. J. en Siegel A. J., 'A Few Modifications to a Standard Library Proposal,
The Modus Quarterly 3, november 1985

Peterson B., 'Modula-2 Library Comments',
The Modus Quarterly 2, april 1985

Verhulst, E., 'An Implementation of the Standard Library for PC',
The Modus Quarterly 3, november 1985

12 Strings

12.1 Inleiding

De module String bevat de bewerkingen voor het verwerken van stringvariabelen. Een string met n tekens wordt gedefinieerd als een tabel tekens. De ondergrens van de index is nul, de bovengrens is n - 1 :

```
TYPE Str = ARRAY [Ø..n - 1] OF CHAR;
```

Alle tekens kunnen aan een stringvariabele worden toegekend. Als de string minder dan n tekens bevat, wordt het laatste geldige teken van de string gevolgd door een afsluitteken. Het afsluitteken is het nulteken, dit wil zeggen het teken met waarde ØC of CHR(Ø). We zullen dit teken verder aanduiden met de identifier endStr :

```
CONST endStr = ØC;
```

12.2 Stringconstanten

Een stringconstante is een reeks tekens die wordt omsloten door de tekens "'" of '"'. De syntaxis voor de stringconstante is :

```
string = '"' { teken } '"'  | "'" { teken } "'"
```

De lengte van een string is het aantal geldige tekens waaruit de string bestaat, het afsluitteken niet inbegrepen. De waarde van een stringconstante met lengte 1 kan door een toekenningsopdracht aan een variabele van het type CHAR worden toegekend. Een stringconstante met lengte één kunnen we beschouwen als een ARRAY [Ø..Ø] OF CHAR.

De lege string is een string met lengte nul en wordt voorgesteld door twee opeenvolgende aanhalingstekens "" of ''.

Voorbeelden :

"" : de lege string;
'a' : een stringconstante met lengte een; past bij een variabele van het type CHAR bij een toekenningsopdracht;
'abc' : een stringconstante met lengte drie.

Soms willen we ook de besturingstekens als een stringconstante voorstellen. Voor deze tekens gebruiken we een octale tekenconstante.

Voorbeeld :
12C : het teken voor het begin een nieuwe regel (linefeed);
15C : het teken voor de verplaatsing naar het begin van de regel.

Een stringconstante met lengte n_1 past tijdens het uitvoeren van een toekenningsopdracht bij een stringvariabele met maximale lengte n_2 als $n_2 >= n_1$.

Voorbeeld :
```
TYPE Str = ARRAY [Ø..9] OF CHAR;
VAR  a   : Str;

a := '';
a := 'a';
a := 'abc';
a := 'abcdefghij'
```

12.3 Strings met variabele lengte

Het aantal geldige tekens van een stringvariabele is variabel. Een stringvariabele heeft dus een variabele lengte. Als de lengte van de string kleiner is dan de maximale lengte, worden de geldige tekens van de string afgesloten met het teken endStr. De tekens na endStr zijn niet gedefinieerd.

Voorbeeld :

```
TYPE Str = ARRAY [Ø..9] OF CHAR;
VAR  a   : Str;
```

De maximale lengte van een stringvariabele met type Str is gelijk aan tien. Als we de opdracht geven

```
a := 'abc'
```

dan hebben de individuele elementen van de variabele a de waarden

```
a[Ø] = 'a'
a[1] = 'b'
a[2] = 'c'
a[3] = endStr
a[4] tot en met a[9] : onbepaald.
```

Als we aan een stringvariabele een constante waarde toekennen waarvan de lengte groter dan of gelijk aan de maximale lengte van de variabele is, komt het endStr-teken niet voor als een element van de variabele. Als de lengte groter is dan de maximale lengte van de variabele, wordt de waarde rechts afgekapt.

Voorbeeld :

```
TYPE Str = ARRAY [Ø..2] OF CHAR;
VAR  a   : Str;
     b   : Str;

a := 'abc';
b := 'abcd'
```

De waarden van de afzonderlijke elementen van a zijn :

```
a[Ø] = 'a'
a[1] = 'b'
a[2] = 'c'
```

De overeenkomstige elementen van b hebben dezelfde waarde. De lengte van de constante is vier, de maximale lengte van b is slechts drie. De waarde van string b wordt rechts afgekapt en is bijgevolg 'abc'.

12.4 Bewerkingen met strings

Voor variabelen van het type string worden de volgende basisbewerkingen gedefinieerd :

1. het toekennen van een waarde aan een string;
2. het samenvoegen van twee strings tot een nieuwe string;
3. het extraheren van een deel van een string;

4. het opzoeken van een patroon in een gegeven string;
5. het vergelijken van twee strings;
6. het wijzigen, toevoegen of verwijderen van een aantal tekens in een string;
7. het bepalen van de lengte van een string;
8. de in- en uitvoer van strings.

De module String verschaft de procedures voor de eerste zeven bewerkingen. De procedures voor de in- en uitvoer van strings behoren tot de modulen SimpleIO, Terminal en Text.

```
DEFINITION MODULE String;
EXPORT QUALIFIED
       (* type *) CompareResult,
       (* proc *) Length,    Assign,    Insert,    Delete,
                  Position,  Substring, Compare,   Concat;

TYPE CompareResult = (less, equal, greater);

PROCEDURE Length(    str     : ARRAY OF CHAR) : CARDINAL;

PROCEDURE Assign(    bron    : ARRAY OF CHAR;
                 VAR doel    : ARRAY OF CHAR;
                 VAR succes  : BOOLEAN);

PROCEDURE Insert(    bron    : ARRAY OF CHAR;
                 VAR doel    : ARRAY OF CHAR;
                     index   : CARDINAL;
                 VAR succes  : BOOLEAN);

PROCEDURE Delete(VAR str     : ARRAY OF CHAR;
                     index   : CARDINAL;
                     lengte  : CARDINAL;
                 VAR succes  : BOOLEAN);

PROCEDURE Position(  patroon : ARRAY OF CHAR;
                     bron    : ARRAY OF CHAR;
                 VAR index   : CARDINAL;
                 VAR gevonden: BOOLEAN);

PROCEDURE Substring( bron    : ARRAY OF CHAR;
                     index   : CARDINAL;
                     lengte  : CARDINAL;
                 VAR doel    : ARRAY OF CHAR;
                 VAR succes  : BOOLEAN);
```

```
PROCEDURE Concat(     bron1   : ARRAY OF CHAR;
                      bron2   : ARRAY OF CHAR;
                  VAR doel    : ARRAY OF CHAR;
                  VAR succes  : BOOLEAN);

PROCEDURE Compare(    string1 : ARRAY OF CHAR;
                      string2 : ARRAY OF CHAR ) :
                   CompareResult;

END String.
```

```
PROCEDURE Length(     str     : ARRAY OF CHAR) : CARDINAL;
```

De functieprocedure Length bepaalt het aantal geldige tekens in de string <str>. De functiewaarde is de lengte van de string.

```
FROM String IMPORT Length;
TYPE Str1Ø  = ARRAY [Ø..9] OF CHAR;
VAR i, j, k : CARDINAL;
    a       : Str1Ø;

i := Length('');
j := Length('abc');
a := '1234';
k := Length(a)
```

Aan de variabelen i, j en k worden respectievelijk de waarden Ø, 3 en 4 toegekend.

```
PROCEDURE Assign(     bron    : ARRAY OF CHAR;
                  VAR doel    : ARRAY OF CHAR;
                  VAR succes  : BOOLEAN);
```

Aan een stringvariabele kunnen we met een toekenningsopdracht de waarde van een stringconstante toekennen. De toekenningsopdracht gebruiken we ook voor stringvariabelen met hetzelfde type.

Voorbeeld :

```
TYPE Str1Ø = ARRAY [Ø..9] OF CHAR;
VAR  a     : Str1Ø;
     b     : Str1Ø;

a := 'abc';
b := a
```

Een waarde met type T_1 past echter niet bij een variabele van een verschillend type T_2 bij het uitvoeren van de toekenningsopdracht. Met de procedure Assign kan de waarde van de string <bron> worden toegekend aan de string <doel>, ook als de typen van <bron> en <doel> van elkaar verschillen. Als de gedeclareerde lengte van de string <doel> kleiner is dan de actuele lengte van de string <bron>, is de waarde van <succes> FALSE en wordt de string <doel> rechts afgekapt.

Voorbeelden :

```
FROM String   IMPORT Assign;
TYPE Str6     = ARRAY [0..5] OF CHAR;
     Str10    = ARRAY [0..9] OF CHAR;
VAR  a        : Str6;
     b        : Str10;
     s        : BOOLEAN;

b := 'b1b2b3';
Assign(b, a, s);
Assign('a1a2a3a4', a, s);
Assign('', a, s);
```

Door de eerste aanroep van Assign wordt de waarde van a 'b1b2b3' en van s TRUE. Door de tweede aanroep kennen we een constante waarde toe aan de variabele a. We kunnen dit ook doen met een toekenningsopdracht. Na de aanroep van de procedure Assign is de waarde van a 'a1a2a3' en van s FALSE. Door de laatste aanroep wordt de waarde van a de nulstring en van s TRUE.

```
PROCEDURE Insert(    bron    : ARRAY OF CHAR;
                 VAR doel    : ARRAY OF CHAR;
                     index   : CARDINAL;
                 VAR succes  : BOOLEAN);
```

Door de procedure Insert wordt een string <bron> toegevoegd in de string <doel>. De string <bron> wordt toegevoegd voor het teken waarvan de positie wordt aangegeven door <index>. De ondergrens voor <index> is nul. Als de positie buiten de actuele lengte van <doel> is, wordt de string <doel> niet gewijzigd en de waarde van <succes> is FALSE. We kunnen deze procedure dus niet gebruiken om twee strings samen te voegen. Als de lengte van de nieuwe waarde groter is dan de gedeclareerde lengte van <doel>, wordt de nieuwe waarde afgekapt en is de waarde van <succes> FALSE.

Voorbeelden :

```
FROM String   IMPORT Insert;
TYPE Str15   = ARRAY [0..14] OF CHAR;
VAR  a, b    : Str15;
     s       : BOOLEAN;

a := 'een drie';
b := 'twee ';
Insert(b, a, 4, s);
```

Door deze opdracht wordt de waarde van a 'een twee drie' en van s TRUE.

```
a := 'een drie vier';
b := 'twee ';
Insert(b, a, 4, s)
```

De lengte van de nieuwe waarde 'een twee drie vier' is achttien en dit is groter dan de maximale lengte van a. De waarde wordt rechts afgekapt en is 'een twee drie v'. De waarde van s is FALSE.

```
a := 'een twee';
b := 'drie';
Insert(b, a, 10, s);
```

De actuele lengte van a is kleiner dan de plaats waar b moet worden toegevoegd. De string a wordt niet gewijzigd en de waarde van s is FALSE.

```
PROCEDURE Delete(VAR str     : ARRAY OF CHAR;
                     index   : CARDINAL;
                     lengte  : CARDINAL;
                 VAR succes  : BOOLEAN);
```

De procedure Delete wist <lengte> tekens van de string <str> vanaf de positie <index>. De ondergrens voor <index> is nul. Als de waarde van <index> buiten de actuele lengte van <str> is, wordt de waarde van <succes> FALSE en wordt <str> niet gewijzigd. Ook als de som van <lengte> en <index> groter is dan de gedeclareerde lengte van <str>, wordt <succes> FALSE en wordt <str> niet gewijzigd.

Voorbeelden :

```
FROM String   IMPORT Delete;
TYPE Str15  = ARRAY [0..14] OF CHAR;
```

```
VAR a      : Str15;
    s      : BOOLEAN;

a := 'een twee drie';
Delete(a, 4, 5, s);
```

Met deze opdracht wordt de waarde van a 'een drie' en van s TRUE.

```
a := 'een twee drie';
Delete(a, 13, 2, s);
```

De positie vanaf waar moet worden gewist valt buiten de actuele lengte van a. De string a wordt niet gewijzigd en s krijgt de waarde FALSE.

```
a := 'een twee drie';
Delete(a, 10, 6, s);
```

De som van de positie en het aantal tekens is groter dan de gedeclareerde lengte van a. De string wordt niet gewijzigd en het resultaat van s is FALSE.

```
PROCEDURE Position(     patroon  : ARRAY OF CHAR;
                        bron     : ARRAY OF CHAR;
                    VAR index    : CARDINAL;
                    VAR gevonden : BOOLEAN);
```

De procedure Position zoekt de string <patroon> op in de string <bron>. Als <patroon> in <bron> voorkomt is de waarde van <index> de positie van het eerste voorkomen van <patroon> in <bron>. De waarde van <gevonden> is dan TRUE. Als <patroon> niet in <bron> voorkomt, is de waarde van <gevonden> FALSE en die van <index> onbepaald. De ondergrens voor <index> is nul.

Voorbeeld :

```
FROM String   IMPORT Position;
TYPE Str15  = ARRAY [0..14] OF CHAR;
VAR  a      : Str15;
     s      : BOOLEAN;
     pos    : CARDINAL;

a := 'een twee een tw';
Position('tw', a, pos, s);
Position('drie', a, pos, s);
```

Na de eerste aanroep van Position is de waarde van pos 4 en van s TRUE. Na de tweede aanroep is de waarde van s FALSE en van pos onbepaald.

```
PROCEDURE Substring(    bron    : ARRAY OF CHAR;
                        index   : CARDINAL;
                        lengte  : CARDINAL;
                    VAR doel    : ARRAY OF CHAR;
                    VAR succes  : BOOLEAN);
```

De procedure Substring kopieert <lengte> tekens van de string <bron> vanaf de positie <index>. Het resultaat komt in de string <doel>. De waarde van <succes> is FALSE in de volgende gevallen :

- de waarde van <index> is groter dan of gelijk aan de actuele lengte van <bron>. De waarde van <doel> wordt niet gewijzigd;
- de waarde van <index> is kleiner dan de actuele lengte van <bron> maar de som van <index> en <lengte> is groter dan de gedeclareerde lengte van <bron>. De waarde van <doel> bevat zoveel mogelijk tekens;
- de waarde van <lengte> is groter dan de gedeclareerde lengte van <doel>. De waarde van <doel> wordt niet gewijzigd.

```
FROM String   IMPORT SubString;
TYPE Str15  = ARRAY [0..14] OF CHAR;
     Str6   = ARRAY [0..5 ] OF CHAR;
VAR  a      : Str15;
     b      : Str6;
     s      : BOOLEAN;

a := 'een twee drie';
Substring(a, 4, 4, b, s);
```

Na de aanroep van de procedure Substring is de waarde van de string b 'twee' en van s TRUE.

```
Substring(a, 13, 2, b, s);
```

De beginpositie valt buiten de actuele lengte van a. De waarde van string b wordt niet gewijzigd en s wordt FALSE.

```
Substring(a, 10, 5, b, s);
```

De som van de beginpositie en van het aantal te kopiëren tekens is vijftien en dus groter dan de gedeclareerde lengte van a. In de

string b worden zoveel mogelijk tekens gekopieerd. De waarde van b wordt 'rie' en de waarde van s wordt FALSE.

```
Substring(a, 3, 8, b, s);
```

Het aantal te kopiëren tekens is 8 en dus groter dan de gedeclareerde lengte van b. De waarde van b wordt niet gewijzigd. De waarde van s wordt FALSE.

```
PROCEDURE Concat(    bron1   : ARRAY OF CHAR;
                     bron2   : ARRAY OF CHAR;
                 VAR doel    : ARRAY OF CHAR;
                 VAR succes  : BOOLEAN);
```

De procedure Concat voegt de strings <bron1> en <bron2> samen tot een nieuwe string <doel>. Als de lengte van de nieuwe string te groot is, wordt de waarde afgekapt en is succes FALSE.

```
FROM String   IMPORT Concat;
TYPE Str10  = ARRAY [0..9] OF CHAR;
VAR  a      : Str10;
     s      : BOOLEAN;
```

```
Concat('een ', 'drie', a, s);
```

Na de aanroep van de procedure Concat is de waarde van a 'een drie' en die van s TRUE.

```
Concat('een twee ', 'drie', a, s);
```

Door deze opdracht wordt de waarde van a 'een twee d', want de maximale lengte van a is 10, en die van s FALSE.

```
PROCEDURE Compare(   string1 : ARRAY OF CHAR;
                     string2 : ARRAY OF CHAR ) :
                 CompareResult;
```

De functieprocedure Compare vergelijkt de strings <string1> en <string2>. Elk teken van de string heeft een rangnummer in de tekenverzameling die door het computersysteem wordt gebruikt (gewoonlijk de ASCII- of EBCDIC-codes).

De strings worden teken voor teken met elkaar vergeleken. Deze vergelijking stopt bij de eerste ongelijkheid of bij het einde van de kortste string.

Voorbeelden :

```
FROM String  IMPORT Compare, CompareResult;

VAR r, s, t : CompareResult;

r := Compare('abc', 'xyz')
s := Compare('abc', 'abcxyz');
t := Compare('', 'abc');
```

De waarde van de variabele r, s en t is steeds 'less'.

```
s := Compare('xyz', 'xyz');
t := Compare('xyz', 'a');
```

De waarde van s is 'equal' en van t is 'greater'.

In- en uitvoer van strings

De bewerkingen voor de in- en uitvoer worden gedefinieerd in de modulen Terminal, Text en SimpleIO. Deze procedures worden in detail beschreven in de paragrafen over de desbetreffende modulen. Voor de volledigheid vermelden we de proceduredefinities :

```
PROCEDURE ReadString(    file  : File;
                     VAR str   : ARRAY OF CHAR;
                     VAR state : FileState);
```

De procedure ReadString leest de string <str> van het bestand <file>. Het lezen van de string eindigt bij het einde van de regel, het einde van het bestand, bij het teken endStr of wanneer het aantal gelezen tekens gelijk is aan de maximale lengte van <str>.

```
PROCEDURE WriteString(    file  : File;
                          str   : ARRAY OF CHAR;
                      VAR state : FileState);
```

De procedure WriteString schrijft de string <str> naar een bestand <file>.

Module Terminal

```
PROCEDURE ReadString(VAR str : ARRAY OF CHAR);
```

De procedure ReadString leest de string <str> van het toetsenbord. De string eindigt met het teken 'einde regel'.

```
PROCEDURE WriteString(   str : ARRAY OF CHAR);
```

De procedure WriteString schrijft de string <str> op het beeldscherm.

Module SimpleIO

```
PROCEDURE ReadString(VAR str : ARRAY OF CHAR);
```

De procedure ReadString leest de string <str> van de standaardinvoer. De procedure werkt zoals Text.ReadString.

```
PROCEDURE WriteString(   str : ARRAY OF CHAR);
```

De procedure WriteString schrijft de string <str> naar de standaarduitvoer. De procedure werkt zoals Text.WriteString.

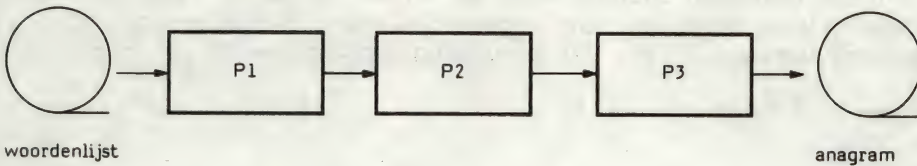
12.5 Toepassingen

We illustreren het gebruik van variabelen met het type string en van de module String met twee uitgewerkte voorbeelden. In het eerste voorbeeld zoeken we alle anagrammen die in een woordenlijst voorkomen. In het tweede voorbeeld definiëren we een bibliotheek-module voor het optellen en aftrekken van gehele getallen die als string worden voorgesteld. Met deze module berekenen we daarna een willekeurig getal uit de rij van Fibonacci.

12.5.1. Anagramprogramma

We geven een lijst met Nederlandse woorden. In deze lijst zoeken we alle woorden die anagrammen vormen. Bijvoorbeeld 'stroop' en 'proost' zijn anagrammen, want deze woorden zijn samengesteld met dezelfde letters, maar in een andere volgorde. We veronderstellen dat elk woord met kleine letters in de woorden-lijst op een afzonderlijke regel is geschreven.

De oplossing van dit probleem organiseren we als een pijpleidingsprogramma (Engels : pipe) met drie stappen. De uitvoer van elk deelprogramma, een stap, is de invoer voor het volgende deelprogramma. We noemen deze stappen voorlopig P_1, P_2 en P_3. De invoer is het bestand 'woordenlijst', de uitvoer het bestand 'anagram' met de verzamelingen van anagrammen. We stellen de verwerking voor door de volgende figuur :



figuur 12.1 Pijpleidingsprogramma 'Anagram'

De eerste stap is het programma FysOpbouw; voor elk woord construeren we de fysieke opbouw. De fysieke opbouw van een woord is een alfabetisch geordende rij van de letters in het woord. Bijvoorbeeld, de fysieke opbouw voor 'spot' en voor 'stop' is 'opst'. Het invoerbestand voor FysOpbouw bevat een woord per regel, bijvoorbeeld :

```
pen
spot
top
nep
stop
pot
```

Elke regel van de uitvoer bevat eerst de fysieke opbouw van een woord en daarna het desbetreffende woord. De fysieke opbouw en het woord worden gescheiden door een spatie. De verwerking van het invoerbestand geeft als resultaat :

```
enp pen
opst spot
opt top
enp nep
opst stop
opt pot
```

In de tweede stap rangschikken we de lijst alfabetisch op het eerste woord. De regels met dezelfde fysieke opbouw worden hierdoor gegroepeerd in opeenvolgende regels. Voor deze bewerking kunnen we een systeemsorteerprogramma gebruiken. We beschikken nu over het bestand :

```
enp nep
enp pen
opst spot
opst stop
opt pot
opt top
```

In de derde en laatste stap voegen we de woorden met dezelfde fysieke opbouw samen tot een regel. We gebruiken hiervoor het programma Verzamel. Met dit programma krijgen we de uitvoer :

```
nep pen
spot stop
pot top
```

We gaan nu de programma's FysOpbouw en Verzamel maken voor de standaardin- en -uitvoer. Als we over elk deelprogramma beschikken, kunnen we het anagramprogramma herleiden tot de opdracht :

```
FysOpbouw < woordenlijst | Sort | Verzamel > anagram
```

De vorm van deze opdracht hangt af van het gebruikte besturingssysteem. In deze opdracht is de invoer voor het programma FysOpbouw het bestand 'woordenlijst'. De uitvoer van FysOpbouw wordt gebruikt als invoer voor programma Sort. De uitvoer van Sort is op zijn beurt de invoer voor het programma Verzamel. Dit laatste programma levert het resultaatbestand 'anagram'.

Het programma FysOpbouw

De invoer voor dit programma is een tekstbestand met één woord per regel. De uitvoer is ook een tekstbestand met per regel de fysieke opbouw van een woord en het woord zelf. De fysieke opbouw en het woord worden gescheiden door een spatie. We formuleren eerst een algemene oplossing in pseudo-taal :

```
lees een woord;
zolang niet einde invoerbestand herhaal
  bepaal de fysieke opbouw;
  schrijf de fysieke opbouw en het woord;
  lees het volgende woord.
```

De fysieke opbouw construeren we als volgt :

```
bepaal de frequentie van de verschillende letters;
schrijf het aantal letters, alfabetisch geordend.
```

Voor het tellen van de letters hebben we reeds in het hoofdstuk 'Modulen' een frequentietabel FreqTab gedefinieerd. Hiervoor geldt de definitiemodule

```
DEFINITION MODULE FreqTab;
EXPORT QUALIFIED TelFrequentie, Frequentie;

PROCEDURE TelFrequentie(teken : CHAR);
PROCEDURE Frequentie(teken : CHAR) : CARDINAL;

END FreqTab.
```

De structuur van de frequentietabel is verborgen in de implementatiemodule. Ook wordt de tabel automatisch geïnitieerd op het ogenblik dat de omgeving die FreqTab invoert, actief wordt. Deze initiëring gebeurt eenmaal.

Voor deze toepassing moet de tabel voor elk woord opnieuw worden ingesteld. We definiëren hiervoor een nieuw abstract datatype FreqTab, waaraan de bewerking InitieerFreqTabel is toegevoegd :

```
DEFINITION MODULE FreqTab;
EXPORT QUALIFIED
  (* proc *) TelFrequentie, Frequentie, InitieerFreqTabel;


PROCEDURE TelFrequentie(teken : CHAR);
PROCEDURE Frequentie(teken : CHAR) : CARDINAL;
PROCEDURE InitieerFreqTabel;

END FreqTab.
```

De tekst wordt gelezen van de standaardinvoer met de bewerkingen ReadString en ReadLn. Het einde van het bestand wordt opgezocht met de functieprocedure EOT. Elke uitvoerregel op de standaarduitvoer wordt gevormd door de uitvoer van de fysieke opbouw, een spatie en het woord zelf. Hiervoor gebruiken we de bewerkingen WriteString en WriteChar. We beginnen een nieuwe regel met WriteLn.

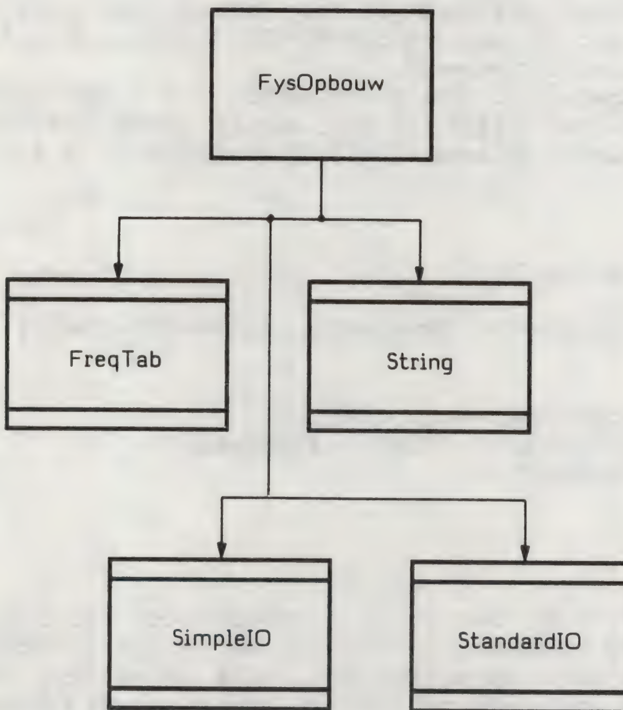
Bij het normale gebruik van de standaardinvoer is de eigenschap 'echo' actief; dit betekent dat elk ingevoerd teken automatisch naar de standaarduitvoer wordt geschreven. Wanneer de standaardinvoer verbonden is met het toetsenbord wordt zo elk ingevoerd teken op het beeldscherm weergegeven.

Als we echter de standaardin- en -uitvoer opnieuw instellen, willen we van deze eigenschap meestal geen gebruik maken. We kunnen de echo besturen met de bewerking SetEchoMode die in de module StandardIO is gedefinieerd. Het argumenttype van deze procedure is het enumeratietype Echo met de waarden (echo, noEcho). Met de opdracht

SetEchoMode(noEcho)

wordt de automatische weergave van de standaardinvoer naar de standaarduitvoer afgezet.

Voor het programma FysOpbouw geldt het abstractieschema :



figuur 12.2 Abstractieschema 'Fysieke Opbouw'

```
MODULE FysOpbouw;
FROM String     IMPORT Length, Concat;
FROM SimpleIO   IMPORT ReadString, ReadLn, EOT,
                       WriteString, WriteLn, WriteChar;
FROM FreqTab    IMPORT InitieerFreqTabel, TelFrequentie,
                       Frequentie;
FROM StandardIO IMPORT SetEchoMode, EchoMode;

CONST maxLengte            = 4Ø;
TYPE  StringIndex          = [Ø..maxLengte - 1];
      Str                  = ARRAY StringIndex OF CHAR;
VAR   s, t                 : Str;
      i                    : StringIndex;
      teken                : ['a'..'z'];
      j                    : CARDINAL;
      succes               : BOOLEAN;
      letterStr            : ARRAY [Ø..Ø] OF CHAR;
BEGIN
SetEchoMode(noEcho);

(* lees de woorden *)
ReadString(s);
WHILE NOT EOT() DO
  ReadLn;
  (* bepaal de frequentie van de letters *)
  InitieerFreqTabel;
  FOR i := Ø TO Length(s) - 1 DO
    TelFrequentie(s[i])
  END;
  (* construeer de fysieke opbouw *)
  t := '';
  FOR teken := 'a' TO 'z' DO
    IF Frequentie(teken) # Ø
    THEN

      FOR j := 1 TO Frequentie(teken) DO
        letterStr[Ø] := teken;
        Concat(t, letterStr, t, succes);
      END
    END
  END;
  (* schrijf de fysieke opbouw en het woord *)
  WriteString(t);
  WriteChar(' ');
  WriteString(s);
  WriteLn;
  (* lees het volgende woord *)
  ReadString(s)
END
END FysOpbouw.
```

We vervolmaken het programma met de implementatie van de module FreqTab :

```
IMPLEMENTATION MODULE FreqTab;

TYPE TabelIndex          = ['a'..'z'];
VAR  frequentieTabel     : ARRAY TabelIndex OF CARDINAL;

PROCEDURE TelFrequentie(teken : CHAR);
BEGIN
INC(frequentieTabel[teken])
END TelFrequentie;


PROCEDURE Frequentie(teken : CHAR) : CARDINAL;
BEGIN
RETURN frequentieTabel[teken]
END Frequentie;


PROCEDURE InitieerFreqTabel;
VAR i : TabelIndex;

BEGIN
FOR i := 'a' TO 'z' DO
  frequentieTabel[i] := 0
END
END InitieerFreqTabel;
END FreqTab.
```

Het programma Verzamel

Het programma Verzamel groepeert op één regel alle woorden met dezelfde fysieke opbouw. De invoer van het programma bestaat uit een aantal regels met per regel de fysieke opbouw, een spatie en een woord. De uitvoer van het programma bestaat uit regels met per regel de woorden met dezelfde fysieke opbouw. De woorden worden gescheiden door een spatie.

Voor elke invoerregel worden de volgende bewerkingen uitgevoerd :

```
splits de regel in de fysieke opbouw van het woord en het woord
zelf;
als de fysieke opbouw verschilt van de vorige
  dan begin een nieuwe uitvoerregel met het woord
  anders voeg het woord toe aan het einde van de huidige
  uitvoerregel.
```

Een regel kunnen we splitsen in de opbouw en het woord zelf wanneer de plaats van het scheidingsteken, een spatie, bekend is. Deze plaats zoeken we met de bewerking Position; het woord en de fysieke opbouw worden bepaald met de bewerking Substring. Als een nieuwe opbouw is gelezen wordt het woord toegekend aan de uitvoerregel met de procedure Assign. Voor dezelfde opbouw wordt het woord aan de huidige uitvoerregel toegevoegd met de procedure Concat.

Voor de in- en uitvoer gebruiken we de bewerkingen ReadString, WriteString en WriteLn. De automatische weergave van de standaardinvoer naar de uitvoer wordt ook hier afgezet door de procedure SetEchoMode.

```
MODULE Verzamel;
FROM String     IMPORT Concat, Length, Assign, CompareResult,
                       Compare, Substring, Position;
FROM SimpleIO   IMPORT ReadString, ReadLn, EOT,
                       WriteString, WriteLn;
FROM StandardIO IMPORT SetEchoMode, EchoMode;

(* definitie van de uitvoerregel *)
CONST maxRegel             = 80;
TYPE  LStr                 = ARRAY [0..maxRegel - 1] OF CHAR;
VAR   uitvoerRegel         : LStr;

(* definitie van de string voor een woord of een opbouw *)
CONST maxLengte            = 40;
TYPE  StringIndex          = [0..maxLengte - 1];
      Str                  = ARRAY StringIndex OF CHAR;
VAR   invoerRegel          : Str;
      woord                : Str;
      plaats               : StringIndex;

(* definitie van een buffer *)
TYPE  HuidigElement        = [0..1];
VAR   element              : HuidigElement;
      buffer               : ARRAY HuidigElement OF Str;

(* overige variabelen *)
VAR   succes, gevonden     : BOOLEAN;
      vergelijkRes         : CompareResult;

BEGIN
(* initieer de uitvoerregel en de echo *)
uitvoerRegel := '';
SetEchoMode(noEcho);
```

```
(* initieer de buffer voor de opbouw *)
element := 1;
buffer [1 - element] := '';

(* lees de eerste invoerregel *)
ReadString(invoerRegel);

(* verwerk de invoerregel *)
WHILE NOT EOT() DO
  ReadLn;

  (* splits de invoerregel in de opbouw en het woord *)
  Position(' ', invoerRegel, plaats, gevonden);
  Substring(invoerRegel, Ø, plaats, buffer[element], succes);
  Substring(invoerRegel, plaats + 1, plaats, woord, succes);

  (* vergelijk de opbouw met de vorige opbouw *)
  vergelijkRes := Compare(buffer[element], buffer[1 - element]);
  IF vergelijkRes = equal
  THEN
    (* voeg het woord toe aan het einde van de uitvoerregel *)
    IF maxRegel - Length(uitvoerRegel) < Length(woord)
    THEN
      WriteString(uitvoerRegel);
      WriteLn;
      Assign(woord, uitvoerRegel, succes)
    ELSE
      Concat(uitvoerRegel, ' ', uitvoerRegel, succes);
      Concat(uitvoerRegel, woord, uitvoerRegel, succes)
    END;

  ELSE
    (* begin met een nieuwe opbouw *)
    IF Length(uitvoerRegel) # Ø
    THEN
      WriteString(uitvoerRegel);
      WriteLn
    END;
    Assign(woord, uitvoerRegel, succes);
    element := 1 - element
  END;

  (* lees de volgende invoerregel *)
  ReadString(invoerRegel)
END;

(* druk de laatste uitvoerRegel *)
WriteString(uitvoerRegel)
END Verzamel.
```

12.5.2. Getalrepresentatie als string

In dit voorbeeld definiëren we enkele bewerkingen voor gehele getallen die als een string worden voorgesteld. Eerst declareren we het type RekenString :

```
CONST maxAantal             = 40;
TYPE  RekenString           = ARRAY [0..maxAantal] OF CHAR;
```

De syntaxis voor de waarde van een getal met type RekenString is :

```
SignificantCijfer =
  '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
cijfer          = '0' | SignificantCijfer
waarde          = ( '+' | '-' )
                  ( SignificantCijfer {cijfer} | '0' )
```

Voorbeeld :

```
VAR a, b : RekenString;

a := '+1234567890';
b := '-10';
```

Het type RekenString kunnen we gebruiken wanneer berekeningen met zeer grote getallen moeten worden uitgevoerd. Het aantal significante cijfers is gelijk aan de constante maxAantal. De eerste positie van een waarde met type RekenString bevat steeds het teken van het getal. Voor de eenvoud beperken we de bewerkingen tot optellen en aftrekken.

```
DEFINITION MODULE RekenStr;
EXPORT QUALIFIED maxAantal, RekenString, TelOp, TrekAf;

CONST maxAantal             = 40;
TYPE  RekenString           = ARRAY [0..maxAantal] OF CHAR;

PROCEDURE TelOp (    s1, s2 : RekenString;
                 VAR res    : RekenString;
                 VAR succes : BOOLEAN);


PROCEDURE TrekAf(   s1, s2 : RekenString;
                 VAR res    : RekenString;
                 VAR succes : BOOLEAN);
END RekenStr.
```

De implementatie van de optel- en aftrekbewerking wordt gerealiseerd met de bewerkingen Concat, Length, Compare, Insert en Delete van de bibliotheekmodule String. We gebruiken ook de standaardprocedures ORD en VAL voor het omzetten van een cijferteken naar een getal en omgekeerd, zodat de elementaire bewerkingen met variabelen met het standaardtype CARDINAL of INTEGER kunnen worden uitgevoerd.
De implementatie bevat drie lokale procedures :

```
PROCEDURE GroterOfGelijk(s1, s2 : RekenString) : BOOLEAN;
```

De invoerargumenten voor deze procedure zijn twee getallen zonder teken. Het resultaat van de functie is TRUE indien voor de getalwaarden van s1 en s2 geldt :

s1 >= s2

```
PROCEDURE TelOpZonderTeken(    s1, s2 : RekenString;
                           VAR res    : RekenString;
                           VAR succes : BOOLEAN);
```

De invoerargumenten van deze procedure zijn twee getallen zonder teken. Het resultaat <res> is de som

s1 + s2

De waarde van <succes> is TRUE, indien de bewerking met succes is uitgevoerd. Bij overloop is de waarde van <succes> FALSE.

```
PROCEDURE TrekAfZonderTeken(    s1, s2 : RekenString;
                            VAR res    : RekenString;
                            VAR succes : BOOLEAN);
```

De invoerargumenten van deze procedure zijn twee getallen zonder teken waarvoor geldt s1 >= s2. Het resultaat <res> is het verschil

s1 - s2

De waarde van <succes> is TRUE indien de bewerking geheel succesvol is uitgevoerd. Bij overloop is de waarde van <succes> FALSE.

```
IMPLEMENTATION MODULE RekenStr;
FROM String    IMPORT Concat, Length, Compare, Insert, Delete,
                      CompareResult;

CONST nul      = 48;  (* ORD('Ø') *)
      nul2     = nul * 2;
TYPE  Index    = [Ø..maxAantal];


(* declaratie van de lokale procedures *)

PROCEDURE GroterOfGelijk(sl, s2 : RekenString) : BOOLEAN;
VAR res : CompareResult;
BEGIN
res := Compare(sl, s2);
RETURN (Length(sl) > Length(s2)) OR
       (Length(sl) = Length(s2)) AND
       ((res = equal) OR (res = greater))
END GroterOfGelijk;


PROCEDURE TelOpZonderTeken(    sl, s2 : RekenString;
                           VAR res    : RekenString;
                           VAR succes : BOOLEAN);
VAR i          : Index;
    lengtel    : Index;
    lengte2    : Index;
    overdracht : [Ø..1];
    cijfer     : [Ø..19];
    cijferStr  : ARRAY [Ø..Ø] OF CHAR;

BEGIN
lengtel := Length(sl);
lengte2 := Length(s2);

(* vul de kortste string op met voorloopnullen *)
IF lengtel < lengte2
THEN
  FOR i := 1 TO lengte2 - lengtel DO
    Concat('Ø', sl, sl, succes)
  END;
  lengtel := lengte2
ELSE
  FOR i := 1 TO lengtel - lengte2 DO
    Concat('Ø', s2, s2, succes)
  END;
  lengte2 := lengtel
END;
```

```
(* tel de twee getalwaarden op *)
(* initiëring *)
res := '';
overdracht := 0;

(* bereken de som *)
FOR i := lengte1 - 1 TO 0 BY -1 DO
  cijfer := ORD(s1[i]) + ORD(s2[i]) + overdracht - nul2;
  IF cijfer > 9
  THEN
    DEC(cijfer,10);
    overdracht := 1
  ELSE
    overdracht := 0
  END;
  cijferStr[0] := VAL(CHAR, cijfer + nul);
  Concat(cijferStr, res, res, succes)
END;

IF overdracht = 1
THEN
  Concat('1', res, res, succes)
END

END TelOpZonderTeken;


PROCEDURE TrekAfZonderTeken(    s1, s2 : RekenString;
                            VAR res    : RekenString;
                            VAR succes : BOOLEAN);

VAR i         : Index;
    lengte1   : Index;
    lengte2   : Index;
    ontlenen  : [0..1];
    cijfer    : [-10..9];
    aantalNul : CARDINAL;
    cijferStr : ARRAY [0..0] OF CHAR;

BEGIN
lengte1 := Length(s1);
lengte2 := Length(s2);

(* vul de kortste string op met voorloopnullen *)
FOR i := 1 TO lengte1 - lengte2 DO
  Concat('0', s2, s2, succes)
END;
```

```
(* initiëring voor het verschil *)
res := '';
ontlenen := Ø;

(* bereken het verschil *)
FOR i := lengtel - 1 TO Ø BY -1 DO
  cijfer := INTEGER(ORD(s1[i])) - INTEGER(ORD(s2[i])) -
            INTEGER(ontlenen);
  IF cijfer < Ø
  THEN
    INC(cijfer, 1Ø);
    ontlenen := 1
  ELSE
    ontlenen := Ø
  END;
  cijferStr[Ø] := VAL(CHAR, cijfer + nul);
  Concat(cijferStr, res, res, succes)
END;

(* wis de niet-significante voorloopnullen *)
(* tel het aantal voorloopnullen *)
aantalNul := Ø;
WHILE (aantalNul < lengtel) AND (res[aantalNul] = 'Ø') DO
  INC(aantalNul)
END;

(* wis dit aantal voorloopnullen *)
IF aantalNul = lengtel
THEN
  res := 'Ø'
ELSE
  Delete(res, Ø, aantalNul, succes)
END

END TrekAfZonderTeken;



(* implementatie van de definitiemodule *)

PROCEDURE TrekAf(    s1, s2 : RekenString;
                 VAR res    : RekenString;
                 VAR succes : BOOLEAN);

CONST plus     = '+';
      min      = '-';
VAR   teken1   : CHAR;
      teken2   : CHAR;
      teken    : CHAR;
```

```
BEGIN
teken1 := s1[0];
teken2 := s2[0];

(* wis het teken *)
Delete(s1, 0, 1, succes);
Delete(s2, 0, 1, succes);

IF teken1 = teken2
THEN
  (* verwerk voor gelijke tekens *)

  (* voer de bewerking uit *)
  IF GroterOfGelijk(s1,s2)
  THEN
    teken := teken1;
    TrekAfZonderTeken(s1, s2, res, succes)
  ELSE
    IF teken1 = plus
    THEN
      teken := min
    ELSE
      teken := plus
    END;
    TrekAfZonderTeken(s2, s1, res, succes)
  END;

  (* voeg het teken toe *)
  IF Compare(res, '0') = equal
  THEN
    Insert(plus, res, 0, succes)
  ELSE
    Insert(teken, res, 0, succes)
  END

ELSE

  (* verwerk voor een verschillend teken *)

  (* voer de bewerking uit *)
  TelOpZonderTeken(s1, s2, res, succes);
  Insert(teken1, res, 0, succes)
END

END TrekAf;
```

```
PROCEDURE TelOp(    s1, s2 : RekenString;
                VAR res    : RekenString;
                VAR succes : BOOLEAN);

VAR teken1     : CHAR;
    teken2     : CHAR;

BEGIN
teken1 := s1[Ø];
teken2 := s2[Ø];

IF teken1 = teken2
THEN
   (* wis het teken *)
  Delete(s1, Ø, 1, succes);
  Delete(s2, Ø ,1, succes);

   (* voer de bewerking uit *)
  TelOpZonderTeken(s1, s2, res, succes);

   (* voeg het teken toe *)
  Insert(teken1, res, Ø succes);

ELSE
   (* pas het teken aan van de tweede term *)
  Delete(s2, Ø, 1, succes);
  Insert(teken1, s2, Ø, succes)

   (* voer de bewerking uit *)
  TrekAf(s1, s2, res, succes)
END
END TelOp
END RekenStr.
```

12.5.3. Programma Fibonacci

We illustreren het gebruik van het type RekenString in de toepassing voor het berekenen van het n-de Fibonacci-getal, met n >= 3. De waarde voor n wordt ingevoerd. De Fibonacci-getallen worden gedefinieerd als

$$f_1 = 0$$

$$f_2 = 1$$

en

$$f_n = f_{n-1} + f_{n-2} \quad \text{voor } n >= 3$$

Voor het berekenen van een willekeurig Fibonacci-getal gebruiken we het type RekenString. Het aantal cijfers in deze representatie is echter beperkt tot RekenStr.maxAantal zodat n naar boven toe is begrensd. Elk Fibonacci-getal bevat hoogstens één cijfer meer dan het voorgaande. Als op een bepaald ogenblik tijdens het berekenen het aantal cijfers van f_n gelijk is aan maxAantal, wordt het programma gestopt.

```
MODULE Fibon;
FROM RekenStr  IMPORT RekenString, TelOp, maxAantal;
FROM String    IMPORT Assign, Length;
FROM SimpleIO  IMPORT ReadCard, WriteString, WriteLn;

VAR f1, f2, fn : RekenString;
    succes     : BOOLEAN;
    teller     : CARDINAL;
    n          : CARDINAL;
BEGIN
f1 := '+0';
f2 := '+1';
fn := '+1';
teller := 3;
WriteString('geef een getal >= 3 : ');
ReadCard(n, succes); WriteLn;
REPEAT
  Assign(f2, f1, succes);
  Assign(fn, f2, succes);
  TelOp(f1, f2, fn, succes);
  INC(teller)
UNTIL (teller = n) OR (Length(fn) = maxAantal + 1);

WriteString('Het '); WriteCard(teller, 4);
WriteString(' getal van Fibonacci is ');
WriteString(fn);WriteLn;
END Fibon.
```

De invoer 185 geeft als resultaat :

+127127879743834334146972278486287885163

Literatuur

Austen G.J.M. en Thomassen H.J., 'UNIX',
Academic Service, Den Haag, 1985

Bentley J.L., 'Programming Pearls',
Communications of the ACM, augustus 1983

Bentley J.L., 'Programming Pearls',
Communications of the ACM, september 1983

Bondy R., 'Modula-2 Standard Library Documentation',
The Modus Quarterly, januari 1985

Boswell F.D., Carmody M.J. en Grove T.R., 'A String Extension For Pascal',
Sigplan, februari 1983

Morrison R., 'The String As a Simple Data Type',
Sigplan

Verhulst E., 'Systeemprogrammatuur en software-ontwikkeling voor microcomputers',
Service, Den Haag, 1984

Oefeningen

1. Voeg aan de module RekenStr de volgende bewerkingen voor getallen met type RekenString toe :
- Produkt;
- Div.

2. Implementeer de bewerking Mod voor het type RekenString.

3. Ontwerp en implementeer een module RomeinsRekenen met de volgende bewerkingen :
- conversie van Romeins getal naar decimaal getal;
- conversie van decimaal naar Romeins getal;
- som van twee Romeinse cijfers;
- verschil van twee Romeinse cijfers.

4. Implementeer een programma Sorteer voor het sorteren van een tekstbestand. Gebruik de standaardin- en uitvoer.

5. Schrijf een procedure ConvertStringToReal voor het omzetten van een string naar een getal met type REAL.

6. Schrijf een procedure ReadCardinal voor de interactieve invoer van getallen met type CARDINAL. De procedure dient alle mogelijke invoerfouten op te vangen. Zorg ook voor de verwerking van de besturingstoetsen, bij voorbeeld 'backspace', 'del', 'esc' of 'tab'.

13 Conversie

13.1 De module Convert

De module Convert bevat de procedures voor het omzetten van gehele getallen van de interne representatie naar strings en omgekeerd.

```
DEFINITION MODULE Convert;
EXPORT QUALIFIED
  (* proc *) IntToStr,  StrToInt, CardToStr, StrToCard,
             NumToStr,  StrToNum;

PROCEDURE IntToStr(   int      : INTEGER;
                  VAR str      : ARRAY OF CHAR;
                      lengte   : CARDINAL;
                  VAR succes   : BOOLEAN);

PROCEDURE CardToStr(  card     : CARDINAL;
                  VAR str      : ARRAY OF CHAR;
                      lengte   : CARDINAL;
                  VAR succes   : BOOLEAN);

PROCEDURE NumToStr(   num      : CARDINAL;
                  VAR str      : ARRAY OF CHAR;
                      basis    : CARDINAL;
                      lengte   : CARDINAL;
                  VAR succes   : BOOLEAN);

PROCEDURE StrToInt(   str      : ARRAY OF CHAR;
                  VAR int      : INTEGER;
                  VAR succes   : BOOLEAN);

PROCEDURE StrToCard(  str      : ARRAY OF CHAR;
                  VAR card     : CARDINAL;
                  VAR succes   : BOOLEAN);
```

```
PROCEDURE StrToNum(    str     : ARRAY OF CHAR;
                   VAR num     : CARDINAL;
                       basis   : CARDINAL;
                   VAR succes  : BOOLEAN);
END Convert.
```

```
PROCEDURE IntToStr(    int     : INTEGER;
                   VAR str     : ARRAY OF CHAR;
                       lengte  : CARDINAL;
                   VAR succes  : BOOLEAN);
```

De procedure IntToStr zet een getal <int> met type INTEGER om in een string <str>. Het aantal tekens van de string wordt gegeven door de parameter <lengte>. Indien de lengte groter is dan het aantal tekens in de voorstelling van het getal, worden vooraan spaties toegevoegd. Indien de lengte te klein is voor de voorstelling van het getal, worden, indien mogelijk, automatisch de nodige tekens toegevoegd. Als de maximale lengte van de string echter te klein is om het getal voor te stellen, is de waarde van <succes> FALSE en is de waarde van <str> onbepaald.

De procedure CardToStr werkt op een soortgelijke manier voor een getal met type CARDINAL. De procedure NumToStr zorgt voor de omzetting van een getal met type CARDINAL in een string. Het getal wordt voorgesteld in het talstelsel met grondtal <basis>.

Voorbeeld :

```
TYPE Str4 = ARRAY [0..3] OF CHAR;
VAR  s    : Str4;
     ok   : BOOLEAN;
```

```
IntToStr(10, s, 4, ok)
```

levert voor s de waarde ' 10' en voor ok de waarde TRUE.

```
IntToStr(100, s, 2, ok)
```

levert voor s de waarde '100' en voor ok de waarde TRUE.

```
IntToStr(12345, s, 3, ok)
```

De waarde van s is onbepaald. De waarde van ok is FALSE.

```
PROCEDURE StrToInt(    str     : ARRAY OF CHAR;
                   VAR int     : INTEGER;
                   VAR succes  : BOOLEAN);
```

De procedure StrToInt zet de string <str> om in een getal met type INTEGER. De omzetting stopt bij het eerste teken dat geen cijfer is. De waarde van succes is FALSE, indien het eerste teken een ander teken is dan een spatie, een cijfer of een '+'- of een '-'-teken. De waarde van succes is ook FALSE, indien de getalwaarde van de string te groot is om intern als een INTEGER te worden voorgesteld.

Evenzo werkt de procedure StrToCard voor het omzetten van een string in een getal met type CARDINAL. Door de procedure StrToNum wordt de getalwaarde van de string <str> met grondtal <basis> omgezet in een getal met type CARDINAL.

13.2 De module ConvertReal

De module ConvertReal bevat de procedures voor het omzetten van getallen met type REAL van de interne representatie naar strings en omgekeerd.

```
DEFINITION MODULE ConvertReal;
EXPORT QUALIFIED RealToStr, StrToReal;

PROCEDURE RealToStr(    real      : REAL;
                    VAR str       : ARRAY OF CHAR;
                        lengte    : CARDINAL;
                        decPlaats : INTEGER;
                    VAR succes    : BOOLEAN);

PROCEDURE StrToReal(    str       : ARRAY OF CHAR;
                    VAR real      : REAL;
                    VAR succes    : BOOLEAN);
END ConvertReal.
```

De procedure RealToStr zet een getal <real> met type REAL om in een string <str>. Het aantal tekens van de string wordt gegeven door de parameter <lengte>. De waarde van <decPlaats> geeft het aantal cijfers aan dat rechts van de decimale punt moet worden weergegeven. Als deze waarde negatief is, wordt het getal weergegeven in de wetenschappelijke notatie; door de waarde nul wordt het getal weergegeven zonder de decimale punt, zoals een getal met type CARDINAL of INTEGER.

De opgegeven waarde voor de lengte is een ondergrens. Als de waarde niet kan worden omgezet met het opgegeven aantal tekens na de decimale punt, wordt het getal weergegeven in de wetenschappelijke notatie met dezelfde lengte. Als ook deze weergave onmogelijk is, worden, indien mogelijk, automatisch tekens toegevoegd om het getal voor te stellen zonder informatieverlies. Als de maximale lengte van de string echter te klein is om het getal voor te stellen, is de waarde van <succes> FALSE en die van <str> onbepaald.

13.3 Rekenen met getallen in een stuk tekst

We definiëren een bibliotheekmodule RekenTekst voor het uitvoeren van rekenkundige bewerkingen op de getallen die in een tekst voorkomen. Deze tekst is bijvoorbeeld aangemaakt met een editor of met een elektronisch werkblad (Engels : spreadsheet).

We onderscheiden de volgende typen getallen : CARDINAL, INTEGER en REAL. De getalwaarden worden geschreven volgens de geldende Modula-2 syntaxis, zodat we de standaardprocedures voor de conversies kunnen gebruiken. Als bewerkingen beschikken we over optellen, aftrekken, vermenigvuldigen, delen en restbepalen. De bewerkingen beperken we tot SelecteerGetal, VoerBewerkingUit en ZetGetalNaarString.

```
DEFINITION MODULE RekenTekst;
EXPORT QUALIFIED
   (* const *) maxRegelLengte,
   (* type  *) RegelIndex, Regel,
               Getal, GetalType,
               Bewerking,
   (* proc  *) SelecteerGetal, VoerBewerkingUit,
               ZetGetalNaarString;

CONST maxRegelLengte    = 80;
TYPE  RegelIndex        = [0..maxRegelLengte - 1];
      Regel             = ARRAY RegelIndex OF CHAR;

TYPE  GetalType         = (cardinal, integer, real);
      Getal             = RECORD
        CASE type : GetalType OF
          cardinal : c : CARDINAL;
        | integer  : i : INTEGER;
        | real     : r : REAL
        END
      END (* Getal *)
```

```
TYPE Bewerking         = (som, verschil, produkt, quotient, rest);

PROCEDURE SelecteerGetal(      regel     : Regel;
                               plaats    : RegelIndex;
                               lengte    : RegelIndex;
                               type      : GetalType;
                           VAR getal     : Getal;
                           VAR succes    : BOOLEAN);

PROCEDURE VoerBewerkingUit(    bewerking : Bewerking;
                               getalA    : Getal;
                               getalB    : Getal;
                           VAR resultaat : Getal);

PROCEDURE ZetGetalNaarString(  getal     : Getal;
                               lengte    : RegelIndex;
                           VAR str       : ARRAY OF CHAR;
                           VAR succes    : BOOLEAN);

END RekenTekst.
```

De procedure SelecteerGetal selecteert in de invoerregel <regel> een string met lengte <lengte> vanaf de positie <plaats>. Deze string wordt omgezet naar een getal met het type <type>. De waarde van <succes> is TRUE als de selectie en de omzetting met succes worden uitgevoerd.

De invoer voor de procedure VoerBewerkingUit bestaat uit de gewenste bewerking <bewerking> en de twee getallen <getalA> en <getalB> waarop de bewerking moet worden uitgevoerd. Het resultaat van de bewerking komt in <resultaat>.

De procedure ZetGetalNaarString zorgt voor het omzetten van <getal> naar een string <str> met lengte <lengte>. De waarde van succes is TRUE als de omzetting met succes wordt uitgevoerd.

Bij de implementatie van deze module houden we rekening met de volgende beperkingen :

1. het type van de eerste operand is gelijk aan het type van de tweede operand;
2. het type van het resultaat is gelijk aan het type van beide operanden.

We gebruiken de bewerkingen van de module Convert en ConvertReal voor het omzetten van strings naar getallen en omgekeerd. We selecteren de string met een getalwaarde uit een regel met de procedure Substring uit module String.

```
IMPLEMENTATION MODULE RekenTekst;
FROM String      IMPORT Substring;
FROM Convert     IMPORT StrToCard, CardToStr,
                        StrToInt,  IntToStr;
FROM ConvertReal IMPORT StrToReal, RealToStr;


PROCEDURE SelecteerGetal(      regel     : Regel;
                               plaats    : RegelIndex;
                               lengte    : RegelIndex;
                               type      : GetalType;
                           VAR getal     : Getal;
                           VAR succes    : BOOLEAN);
VAR str : ARRAY [0..20] OF CHAR;
BEGIN
Substring(regel, plaats, lengte, str, succes);
IF NOT succes THEN RETURN END;

getal.type := type;

CASE getal.type OF
  cardinal : StrToCard(str, getal.c, succes);
| integer  : StrToInt (str, getal.i, succes);
| real     : StrToReal(str, getal.r, succes)
END
END SelecteerGetal;

PROCEDURE VoerBewerkingUit(    bewerking : Bewerking;
                               getalA    : Getal;
                               getalB    : Getal;
                           VAR resultaat : Getal);
BEGIN
resultaat.type := getalA.type;
CASE resultaat.type OF
  cardinal :
    CASE bewerking OF
      som       : resultaat.c := getalA.c + getalB.c
    | verschil  : resultaat.c := getalA.c - getalB.c
    | produkt   : resultaat.c := getalA.c * getalB.c
    | quotient  : resultaat.c := getalA.c DIV getalB.c
    | rest      : resultaat.c := getalA.c MOD getalB.c
    END
```

```
| integer :
    CASE bewerking OF
      som      : resultaat.i := getalA.i + getalB.i
    | verschil : resultaat.i := getalA.i - getalB.i
    | produkt  : resultaat.i := getalA.i * getalB.i
    | quotient : resultaat.i := getalA.i DIV getalB.i
    | rest     : resultaat.i := getalA.i MOD getalB.i
    END
| real :
    CASE bewerking OF
      som      : resultaat.r := getalA.r + getalB.r
    | verschil : resultaat.r := getalA.r - getalB.r
    | produkt  : resultaat.r := getalA.r * getalB.r
    | quotient : resultaat.r := getalA.r / getalB.r
    | rest     : (* niet gedefinieerd *)
END
END VoerBewerkingUit;

PROCEDURE ZetGetalNaarString(    getal     : Getal;
                                 lengte    : RegelIndex;
                             VAR str       : ARRAY OF CHAR;
                             VAR succes    : BOOLEAN);
CONST decPlaats = 2;
BEGIN
CASE getal.type OF
  cardinal : CardToStr(getal.c, str, lengte, succes)
| integer  : IntToStr (getal.i, str, lengte, succes);
| real     : RealToStr(getal.r, str, lengte, decPlaats, succes)
END
END ZetGetalNaarString;

END RekenTekst.
```

We kunnen de module RekenTekst gebruiken in een editor met rekenfaciliteiten, in een elektronisch werkblad of voor het verwerken van een tekstbestand waarin de getallen op een vaste positie in een regel voorkomen.

Voorbeeld :
De invoer van een programma bestaat uit een aantal regels. Elke regel bevat een produktomschrijving, een hoeveelheid en een eenheidsprijs. De getalwaarden voor 'hoeveelheid' en 'prijs' worden vermeld in de dertigste en de vijftigste kolom. De ruimte voor een getal is telkens vijf tekens.

Elke uitvoerregel van het programma bestaat uit de invoerregel aangevuld met het totaalbedrag per regel. Dit bedrag, het product van 'hoeveelheid' en 'prijs', wordt toegevoegd vanaf de zestigste kolom, tien tekens lang.

```
invoerbestand :          aantal          prijs
artikel_1                xxxxx           xxxxx
artikel_2                xxxxx           xxxxx

uitvoerbestand :                                   totaal
artikel_1                xxxxx           xxxxx     xxxxxxxxxx
artikel_2                xxxxx           xxxxx     xxxxxxxxxx
```

```
MODULE BerekenTotaal;
FROM RekenTekst IMPORT Regel, Getal, GetalType, Bewerking,
                       SelecteerGetal, VoerBewerkingUit,
                       ZetGetalNaarString;
FROM SimpleIO   IMPORT WriteString, WriteLn,
                       ReadString, ReadLn, EOT;
FROM String     IMPORT Concat;
FROM StandardIO IMPORT SetEchoMode, EchoMode;

VAR r          : Regel;
    a, b, c    : Getal;
    succes     : BOOLEAN;
    str        : ARRAY [Ø..2Ø] OF CHAR;

BEGIN
SetEchoMode(noEcho);

ReadString(r);
WHILE NOT EOT() DO
  ReadLn;
  SelecteerGetal(r, 3Ø, 5, cardinal, a, succes);
  SelecteerGetal(r, 5Ø, 5, cardinal, b, succes);
  VoerBewerkingUit(produkt, a, b, c);
  ZetGetalNaarString(c, 1Ø, str, succes);
  Concat(r, '    ', r, succes);
  Concat(r, str, r, succes);
  WriteString(r);
  WriteLn;
  ReadString(r)
END
END BerekenTotaal.
```

Oefeningen

1. Voeg het type RekenString toe aan de module RekenTekst.

2. Wijzig de definitie en de implementatie van module RekenTekst zodat ook bewerkingen met gemengde typen kunnen worden verwerkt.

14 Wiskundige functies

De module MathLib

De module MathLib bevat de standaardfuncties Sqrt, Exp, Ln, Sin, Cos, Arctan, Entier en Power. De argumenten voor de goniometrische functies worden uitgedrukt in radialen. De functie Entier zet het gehele deel van een getal met type REAL om in een getal met type INTEGER.

Bij een ongeldig argument, bijvoorbeeld een negatieve waarde voor het argument van de vierkantswortel of overloop, wordt de verwerking van het programma gestopt.

```
DEFINITION MODULE MathLib;
EXPORT QUALIFIED
  (* proc *) Sqrt, Exp, Ln, Sin, Cos, Arctan, Entier, Power;

PROCEDURE Sqrt   (real : REAL) : REAL;

PROCEDURE Exp    (real : REAL) : REAL;

PROCEDURE Ln     (real : REAL) : REAL;

PROCEDURE Sin    (real : REAL) : REAL;

PROCEDURE Cos    (real : REAL) : REAL;

PROCEDURE Arctan (real : REAL) : REAL;

PROCEDURE Entier (real : REAL) : INTEGER;

PROCEDURE Power  (real : REAL;
                   exp  : REAL) : REAL;

END MathLib.
```

15 Het werkstation

15.1 Module Terminal

De module Terminal verschaft op eenvoudige en directe wijze toegang tot het werkstation. Het systeem voor bestandsbeheer wordt hierbij gepasseerd. Hierdoor worden het rendement en de snelheid van de in- en uitvoer naar het werkstation vergroot. Daarentegen zijn sommige faciliteiten van het bestandsbeheer niet beschikbaar zoals de instelling voor de automatische weergave van de invoer (echo), de 'redirection' van de standaardin- en -uitvoer en het terugzetten van het laatste gelezen teken in het invoerbestand (UndoRead).

De module wordt vooral toegepast in scherm-georiënteerde programma's en in systemen met ingebouwde logica (Engels : embedded systems). De module is ontworpen voor een tekstscherm : dat wil zeggen elke regel bestaat uit n regels (vaak 24) met maximaal k posities (vaak 8Ø).

```
DEFINITION MODULE Terminal;
EXPORT QUALIFIED
  (* proc *) ReadChar, ReadString, CondRead,
             WriteChar, WriteString, WriteLn,
             NumRows, NumCols, GotoRowCol,
             EraseScreen, EraseToEOL, EraseToEOS;


PROCEDURE ReadChar    (VAR ch     : CHAR);

PROCEDURE ReadString  (VAR str    : ARRAY OF CHAR);

PROCEDURE CondRead    (VAR ch     : CHAR;
                       VAR succes : BOOLEAN);


PROCEDURE WriteChar   (    ch     : CHAR);

PROCEDURE WriteString (    str    : ARRAY OF CHAR);
```

```
PROCEDURE WriteLn;

PROCEDURE NumRows     () : CARDINAL;

PROCEDURE NumCols     () : CARDINAL;

PROCEDURE GotoRowCol  (    rij   : CARDINAL;
                           kolom : CARDINAL);


PROCEDURE EraseScreen;

PROCEDURE EraseToEOL;

PROCEDURE EraseToEOS;

END Terminal.
```

```
PROCEDURE ReadChar    (VAR ch     : CHAR);
```

De procedure ReadChar leest een teken <ch> van het toetsenbord. De procedure wacht tot het teken is ingevoerd. Het teken wordt weergegeven op het beeldscherm. De procedure interpreteert de ingevoerde tekens niet. Besturingstekens en de tekens voor het einde van de regel of voor het einde van een bestand dienen in het programma zelf te worden verwerkt.

```
PROCEDURE ReadString  (VAR str    : ARRAY OF CHAR);
```

De procedure ReadString leest een string <str> van het toetsenbord. De leesopdracht eindigt bij het teken 'einde regel', 'einde bestand' of als de string geheel met tekens is gevuld. De string wordt op het scherm weergegeven.

```
PROCEDURE CondRead   (VAR ch      : CHAR;
                      VAR succes  : BOOLEAN);
```

De procedure CondRead leest een teken <ch> van het toetsenbord. De procedure wacht <u>niet</u> tot het teken is ingevoerd. Als een teken is ingevoerd is de waarde van <succes> TRUE. Anders is de waarde van <succes> FALSE en die van <ch> onbepaald. Het teken wordt <u>niet</u> weergegeven op het beeldscherm. De procedure interpreteert de ingevoerde tekens niet.

Besturingstekens en de tekens voor het einde van de regel of voor het einde van een bestand dienen in het programma zelf te worden verwerkt.

```
PROCEDURE WriteChar   (    ch     : CHAR);
```

De procedure WriteChar schrijft het teken <ch> naar het beeldscherm.

```
PROCEDURE WriteString (    str    : ARRAY OF CHAR);
```

De procedure WriteString schrijft de string <str> naar het beeldscherm.

```
PROCEDURE WriteLn;
```

De procedure WriteLn gaat naar een nieuwe regel op het beeldscherm.

```
PROCEDURE NumRows     () : CARDINAL;
PROCEDURE NumCols     () : CARDINAL;
```

Met de procedures NumRows en NumCols kunnen we de afmetingen van het scherm opvragen. De functieprocedure NumRows geeft het aantal regels, de functieprocedure NumCols het aantal tekens per regel (kolommen).

```
PROCEDURE GotoRowCol  (    regel : CARDINAL;
                           kolom : CARDINAL);
```

De procedure GotoRowCol verplaatst de cursor naar de regel <regel> en naar kolom <kolom>. De regels worden genummerd van 1 tot en met NumRows, de kolommen van 1 en met tot NumCols. De coördinaten van de linkerbovenhoek van het scherm zijn (1, 1).

```
PROCEDURE EraseScreen;
```

De procedure EraseScreen wist het scherm.

```
PROCEDURE EraseToEOL;
```

De procedure EraseToEOL wist alle tekens tot het einde van de regel.

PROCEDURE EraseToEOS;

De procedure EraseToEOS wist alle tekens tot het einde van het scherm.

15.2 Een uitbreiding voor de module Terminal

De bewerkingen van de module Terminal zijn voor de meeste scherm-georiënteerde programma's te beperkt. Het werkstation van een computersysteem is meestal samengesteld uit een beeldscherm en een toetsenbord. Hieraan kan nog andere apparatuur voor de in- en uitvoer worden toegevoegd; een muis, een lichtpen of het gebruik van 'spraak'. Voor de besturing van het beeldscherm zijn reeds enkele bewerkingen in de module Terminal gedefinieerd. We kunnen aan deze bewerkingen nog andere toevoegen voor de cursorbesturing en voor het instellen van de manier van afdrukken. Voor deze nieuwe bewerkingen definiëren we de module Beeldscherm. Voor de aansturing van het toetsenbord beschikt de module Terminal alleen over de bewerkingen ReadChar, CondRead en ReadString. Het toetsenbord is echter samengesteld uit verschillende toetsengroepen : programmeerbare functietoetsen, toetsen voor de cursorbesturing, cijfer- en lettertoetsen en besturingstoetsen. We definiëren ook nog een module Toetsenbord met enkele bewerkingen voor het aansturen van deze toetsengroepen.

De module Beeldscherm

Voor het aansturen van het beeldscherm onderscheiden we twee groepen bewerkingen : de cursorbesturing en het instellen van de 'afdruk'-eigenschappen.

De cursorbesturing

Voor de cursorbesturing beschikt de module Terminal reeds over de bewerkingen NumRows, NumCols en GotoRowCol. We voegen hieraan bewerkingen toe voor het opvragen, het bewaren en het wijzigen van de cursorpositie.
Het scherm wordt gedefinieerd als :

```
CONST maxRegel = 25; (* afhankelijk van NumRows *)
      maxKolom = 80; (* afhankelijk van NumCols *)
TYPE  Regel    = [1..maxRegel];
      Kolom    = [1..maxKolom];
```

```
Plaats   = RECORD
  regel : Regel;
  kolom : Kolom
END;
```

De procedure GeefPositieCursor geeft de huidige plaats van de cursor :

```
PROCEDURE GeefPositieCursor(VAR plaats : Plaats);
```

De procedure RedCursor registreert de huidige plaats van de cursor. Met de procedure HerstelCursor wordt de cursor teruggezet op de plaats die vooraf met de laatst uitgevoerde aanroep van RedCursor werd geregistreerd.

```
PROCEDURE RedCursor;

PROCEDURE HerstelCursor;
```

Voor het verplaatsen van de cursor in de verschillende richtingen definiëren we de volgende procedures :

```
PROCEDURE VerplaatsLinks;

PROCEDURE VerplaatsRechts;

PROCEDURE VerplaatsBoven;

PROCEDURE VerplaatsOnder;
```

Op de rand van het beeldscherm heeft de aanroep voor een verplaatsing buiten het beeldscherm geen effect.

Het instellen van het scherm voor de weergave van tekens

Bij de weergave van de tekens op het beeldscherm kunnen allerlei zaken worden ingesteld. De belangrijkste zijn : het onderstrepen, de intensiteit, het aanduiden van verwijderde tekens, de keuze van een lettertype (fonttype), de keuze van de kleur voor de voor- of achtergrond. Elk van deze eigenschappen wordt in werking gezet met een daartoe bestemde bewerking. Zolang een eigenschap actief is, worden alle tekens, ook de spatie, met de desbetreffende eigenschap weergegeven. De eigenschap blijft actief tot ze wordt afgezet. Verschillende weergave-eigenschappen kunnen tegelijk actief zijn.

Het onderstrepen

We kunnen niet, enkel of dubbel onderstrepen. Slechts een van deze eigenschappen is op een bepaald ogenblik actief. Het activeren van een bepaalde instelling stopt automatisch de vorige. We definiëren het enumeratietype Onderstreping :

```
TYPE Onderstreping = (geenOnderstreping, enkeleOnderstreping,
                      dubbeleOnderstreping);
```

en de bewerking :

```
PROCEDURE SelecteerOnderstreping(onderstreping : Onderstreping);
```

De systeemgekozen waarde voor onderstreping is 'geenOnderstreping'.

De intensiteit

De tekst op het beeldscherm kan met een wisselende intensiteit worden weergegeven zodat het contrast met de achtergrond meer of minder wordt beklemtoond. We onderscheiden de intensiteiten normaal, laag en hoog :

```
TYPE Intensiteit = (normaal, laag, hoog);
```

Slechts een van de drie is op een bepaald ogenblik actief. De keuze van een intensiteit stopt automatisch de huidige instelling. We definiëren de bewerking :

```
PROCEDURE SelecteerIntensiteit(intensiteit : Intensiteit);
```

De systeemgekozen waarde voor intensiteit is 'normaal'.

Het aanduiden van verwijderde tekens

Met deze eigenschap worden de tekens aangeduid die in de tekst mogen worden verwijderd. De wijze waarop deze verwijderde tekens worden aangegeven is afhankelijk van het systeem.

```
TYPE Verwijderd = (nietVerwijderd, teVerwijderen);
PROCEDURE SelecteerVerwijderd(verwijderd : Verwijderd).
```

De systeemgekozen waarde is 'nietVerwijderd'.

De keuze van de voor- of achtergrondkleur

Op een kleurenscherm kan de voor- of achtergrondkleur worden ingesteld. We onderscheiden de volgende kleuren :

```
TYPE Kleur = (zwart, rood, groen, geel, blauw, magenta,
              cyaan, wit);
```

De kleuren worden ingesteld met de procedures

```
PROCEDURE SelecteerVoorgrond  (kleur : Kleur);

PROCEDURE SelecteerAchtergrond(kleur : Kleur);
```

De keuze van de lettertype

Voor een tekstscherm is momenteel meestal slechts één enkel lettertype beschikbaar. Het systeemgekozen lettertype noemen we het primaire lettertype, de overige de alternatieve lettertypen. Het aantal van deze alternatieve lettertypen wordt door het systeem bepaald. We definiëren het type Lettertype en de procedure SelecteerLettertype :

```
CONST maxLettertype = 9;
      (* bij voorbeeld negen alternatieve lettertypen *)
TYPE  Lettertype    = [Ø..maxLettertype];

PROCEDURE SelecteerLettertype(familie : Lettertype);
```

Het herstellen van de systeemgekozen toestand

De verschillende eigenschappen kunnen worden ingesteld met de hiervoor genoemde bewerkingen. Met de procedure SelecteerSysteem worden de systeemgekozen eigenschappen ingesteld : normale intensiteit, geen onderstreping, niet verwijderd en het primaire lettertype.

```
PROCEDURE SelecteerSysteem;
```

We definiëren nu de volledige definitiemodule voor het aansturen van het scherm. Sommige van de eigenschappen kunnen waarschijnlijk niet op de gebruikte beeldschermen worden toegepast. We behouden echter de definitiemodule en we vangen deze beperkingen op in de implementatiemodule.

```
DEFINITION MODULE Beeldscherm;
EXPORT QUALIFIED
  (* cursorbesturing *)
  (* type *) Regel, Kolom, Plaats,
  (* proc *) GeefPositieCursor, RedCursor, HerstelCursor,
             VerplaatsLinks, VerplaatsRechts, VerplaatsBoven,
             VerplaatsOnder,

  (* weergave-eigenschappen *)
  (* const *) maxLettertype,
  (* type *) Onderstreping, Intensiteit, Verwijderd, Kleur,
             Lettertype,
  (* proc *) SelecteerOnderstreping, SelecteerIntensiteit,
             SelecteerVerwijderd, SelecteerVoorgrond,
             SelecteerAchtergrond, SelecteerLettertype,
             SelecteerSysteem;

CONST maxRegel = 25; (* afhankelijk van Terminal.NumRows *)
      maxKolom = 80; (* afhankelijk van Terminal.NumCols *)
TYPE  Regel    = [1..maxRegel];
      Kolom    = [1..maxKolom];
      Plaats   = RECORD
        regel : Regel;
        kolom : Kolom
      END;

PROCEDURE GeefPositieCursor(VAR plaats : Plaats);
PROCEDURE RedCursor;
PROCEDURE HerstelCursor;
PROCEDURE VerplaatsLinks;
PROCEDURE VerplaatsRechts;
PROCEDURE VerplaatsBoven;
PROCEDURE VerplaatsOnder;


TYPE Onderstreping = (geenOnderstreping, enkeleOnderstreping,
                      dubbeleOnderstreping);
PROCEDURE SelecteerOnderstreping(onderstreping : Onderstreping);

TYPE Intensiteit  = (normaal, laag, hoog);
PROCEDURE SelecteerIntensiteit(intensiteit : Intensiteit);

TYPE Verwijderd    = (nietVerwijderd, teVerwijderen);
PROCEDURE SelecteerVerwijderd(verwijderd : Verwijderd);
```

```
TYPE Kleur        = (zwart, rood, groen, geel, blauw, magenta,
                     cyaan, wit);
PROCEDURE SelecteerVoorgrond  (kleur : Kleur);
PROCEDURE SelecteerAchtergrond(kleur : Kleur);


CONST maxLettertype = 9;
      (* bijvoorbeeld negen alternatieve lettertypen *)
TYPE  Lettertype    = [Ø..maxLettertype];
PROCEDURE SelecteerLettertype(familie : Lettertype);


PROCEDURE SelecteerSysteem;
END Beeldscherm.
```

De implementatie van de module Beeldscherm is sterk afhankelijk van het toegepaste computersysteem en de daaraan gekoppelde beeldschermen. Rekening houdend met deze sterke afhankelijkheid, vermelden we hier geen specifieke implementatie.

De module Toetsenbord

Bij de meeste toetsenborden onderscheiden we de functietoetsen, de toetsen voor de cursorbesturing, het numeriek blok en het hoofdtoetsenbord met besturingstoetsen. De betekenis van de functietoetsen en de besturingstoetsen wordt meestal bepaald door de programma's die op het systeem draaien. We definiëren nu een module Toetsenbord voor het aansturen van de functietoetsen en de toetsen voor de cursorbesturing.

We veronderstellen dat de functietoetsen op het toetsenbord worden aangeduid met de symbolen 'F1', 'F2' en zo verder tot en met 'F1Ø'. In de programma's willen we ook de symbolen 'F1', 'F2' en verder gebruiken. We definiëren het type Functie als :

```
TYPE Functie = (F1, F2, F3, F4, F5, F6, F7, F8, F9, F1Ø);
```

Met de functieprocedure FunctieToets wordt een functietoets gelezen. Deze functie tast het toetsenbord af tot een functietoets is ingevoerd.

```
PROCEDURE FunctieToets() : Functie;
```

Voor de cursorbesturing hebben we de toetsen voor het verplaatsen naar links, rechts, boven en onder nodig. Hiervoor definiëren we het type

```
TYPE RichtingPijl = (links, rechts, boven, onder);
```

Met de functieprocedure Pijl wordt de invoer van een toets voor de cursorbesturing gelezen. Deze functie tast het toetsenbord af tot een pijl is ingevoerd.

```
PROCEDURE Pijl() : RichtingPijl;
```

```
DEFINITION MODULE Toetsenbord;
EXPORT QUALIFIED
  (* type *) Functie, RichtingPijl,
  (* proc *) FunctieToets, Pijl;

TYPE Functie = (F1, F2, F3, F4, F5, F6, F7, F8, F9, F10);
PROCEDURE FunctieToets() : Functie;

TYPE RichtingPijl = (links, rechts, boven, onder);
PROCEDURE Pijl() : RichtingPijl;

END Toetsenbord.
```

De implementatie van deze module is zeer afhankelijk van het systeem en wordt bijgevolg hier niet opgenomen. We gebruiken de module in het hoofdstuk over de aanroep van programma's voor de realisatie van een menugestuurd systeem.

Literatuur

ECMA, European Computer Manufacturers Association, 'Standard ECMA-101', september 1985

ISO 6429 Standard

Oefeningen

1. Implementeer de module Beeldscherm.

2. Implementeer de module Toetsenbord.

3. Definieer en implementeer een module KettingDrukker voor het aansturen van een drukker voor kettingformulieren.

4. Definieer en implementeer een module LetterwielDrukker voor het aansturen van een letterwieldrukker voor afzonderlijke bladzijden.

5. Definieer en implementeer een module LaserDrukker voor het aansturen van een laserdrukker voor afzonderlijke bladzijden.

16 Werken met bestanden

16.1 Inleiding

De bibliotheek maakt onderscheid tussen twee typen van bestanden : binaire bestanden en tekstbestanden. De structuur van een binair bestand wordt volledig bepaald door het programma dat met het bestand werkt. Een tekstbestand daarentegen is samengesteld uit strings, regels en tekstbesturingstekens. Binaire bestanden worden behandeld met de bewerkingen van de module Binary. Tekstbestanden kunnen we op twee manieren manipuleren : de meeste mogelijkheden worden geboden door de bewerkingen van de module Text. Met deze bewerkingen programmeren we de foutbehandeling en de 'redirection' van bestanden, en ook het aanmaken van een logbestand. De procedures van de module SimpleIO zijn eenvoudiger voor de programmeur dan de overeenkomstige procedures van de module Text. Logische fouten kunnen we echter niet opvangen met de bewerkingen van SimpleIO. Deze fouten onderbreken het programma.

16.2 Fatale en niet-fatale fouten

In de bibliotheek wordt onderscheid gemaakt tussen twee typen van fouten : fatale en niet-fatale fouten. Een niet-fatale fout is een fout die door de programmeur niet kan worden voorzien, bijvoorbeeld het openen van een bestand dat niet bestaat. We kunnen deze fout tijdens het uitvoeren ontdekken indien we de toestand van het bestand onderzoeken. Het programma kan dan eventueel op een gecontroleerde wijze worden gestopt; in een interactieve omgeving wordt de fout aan de gebruiker meegedeeld, zodat deze de gepaste maatregelen kan treffen.

Als een niet-fatale fout optreedt en als daarna op de verkeerde gegevensstructuur opnieuw een bewerking wordt uitgevoerd, heeft dit een fatale fout tot gevolg. Het programma wordt door het systeem gestopt zonder dat de gebruiker nog kan ingrijpen.

Een ander voorbeeld van een niet-fatale fout is een DeviceError tijdens het schrijven van een bestand. DeviceError wijst op een fout in de apparatuur voor de in- en uitvoer. We kunnen de informatie over deze fout gebruiken om het programma te laten stoppen. Als we echter met het apparaat verder willen schrijven geeft dit aanleiding tot een fatale fout en het programma wordt gestopt. Soms willen we ondanks de aanwezigheid van fouten toch verder werken met een bestand. Voor deze toepassingen kan de toestand van een bestand worden hersteld met de bewerking ResetState. Als we deze procedure aanroepen na het vinden van een fout, hebben we verder toegang tot het bestand. In dit geval genereert het systeem geen fatale fout.

16.3 De module Files

De module Files wordt steeds samen met de module Text of de module Binary gebruikt. In deze module wordt het verborgen type File gedefinieerd. De module bevat de bewerkingen voor het openen en sluiten van bestanden, het opvragen en wijzigen van de toestand van een bestand en het opvragen van de bestandsnaam.

```
DEFINITION MODULE Files;
EXPORT QUALIFIED
  (* type *) File,        FileState,
             BinTextMode, ReadWriteMode, ReplaceMode,
  (* proc *) Open,        Create,        Close,        Remove,
             Reset,       Rewrite,       Truncate,     Flush,
             EOF,         State,         ResetState,
             GetFileName;
TYPE
    File;
    BinTextMode   = (binMode, textMode);
    ReadWriteMode = (readOnly, readWrite, appendOnly);
    ReplaceMode   = (noReplace, replace);
    FileState     = (ok,
                     nameError, noFile, existingFile,
                     deviceError, noMoreRoom, accessError,
                     notOpen, endError, outsideFile,
                     otherError);

PROCEDURE Open    (VAR file      : File;
                       naam      : ARRAY OF CHAR;
                       binTekst  : BinTextMode;
                       writeMode : ReadWriteMode;
                   VAR status    : FileState);
```

```
PROCEDURE Create   (VAR file      : File;
                        naam      : ARRAY OF CHAR;
                        binTekst  : BinTextMode;
                        vervMode  : ReplaceMode;
                    VAR status    : FileState);

PROCEDURE Close    (VAR file      : File;
                    VAR status    : FileState);

PROCEDURE Remove   (VAR file      : File;
                    VAR status    : FileState);

PROCEDURE Reset    (    file      : File;
                    VAR status    : FileState);

PROCEDURE Rewrite  (    file      : File;
                    VAR status    : FileState);

PROCEDURE Truncate(     file      : File;
                    VAR status    : FileState);

PROCEDURE Flush    (    file      : File;
                    VAR status    : FileState);

PROCEDURE EOF      (    file      : File) : BOOLEAN;

PROCEDURE State    (    file      : File) : FileState;

PROCEDURE ResetState(   file      : File;
                    VAR status    : FileState);

PROCEDURE GetFileName(  file      : File;
                    VAR naam      : ARRAY OF CHAR;
                    VAR status    : FileState);

END Files.
```

16.4 De toegang tot een bestand

De toegang tot een bestand wordt bepaald door de eigenschappen BinTextMode, ReadWriteMode en ReplaceMode. Elk van deze eigenschappen is gedefinieerd als een enumeratietype.

BinTextMode = (binMode, textMode)

De eigenschap binMode bepaalt dat het bestand slechts toegankelijk is met de bewerkingen voor binaire bestanden. We gebruiken dus steeds de bewerkingen van de module Binary voor het uitvoeren van de lees- en schrijfopdrachten. Met de eigenschap textMode is het bestand ook toegankelijk met de bewerkingen van de module Text.

ReadWriteMode = (readOnly, readWrite, appendOnly);

De eigenschap readOnly bepaalt dat van het bestand alleen kan worden gelezen. Met de eigenschap readWrite kunnen we een bestand zowel lezen als schrijven. Met de eigenschap appendOnly kunnen alleen nieuwe gegevens aan het bestand worden toegevoegd.

ReplaceMode = (noReplace, replace);

Bij de creatie van een bestand wordt de inhoudstabel van het externe medium onderzocht. De eigenschap noReplace betekent dat een bestaand bestand niet door een nieuw bestand mag worden vervangen. Met de eigenschap replace wordt een bestaand bestand vervangen door een nieuw bestand.

16.5 De toestand van een bestand

Veel bewerkingen met bestanden beïnvloeden de toestand van het bestand. De toestand weerspiegelt aldus het resultaat van de laatste bewerking die op het bestand is uitgevoerd. De toestand kan steeds worden opgevraagd met de functie State en wordt ook als variabele-parameter doorgegeven bij de aanroep van alle procedures die de toestand van het bestand kunnen wijzigen. Als we een bestand waarvan de toestand niet in orde is willen manipuleren, wordt de verwerking gestopt. Elke onverwachte onderbreking van een programma kunnen we voorkomen als de toestand van een bestand na elke mogelijke wijziging wordt onderzocht en eventueel wordt gewijzigd. De betekenis van de waarden voor de toestand van een bestand zijn :

ok

De laatste bewerking is met succes uitgevoerd. Het bestand is in orde voor de volgende bewerking.

De volgende toestanden kunnen optreden bij de creatie of bij het openen van een bestand :

nameError

De bestandsnaam, die wordt doorgegeven aan de procedure om het bestand te openen, is niet juist geformuleerd.

ExistingFile

Het bestand met de gegeven naam bestaat reeds.

De volgende toestanden kunnen voorkomen bij het openen of bij het bewerken van een bestand :

deviceError

Er is een fout ontdekt in de in- en uitvoerapparatuur.

noMoreRoom

In de inhoudstabel of op het externe medium zelf is onvoldoende plaats beschikbaar om nieuwe gegevens te registreren.

accessError

De toegang tot het bestand stemt niet overeen met de eigenschappen waarmee het bestand is geopend, bijvoorbeeld binair-tekst, read-write.

De volgende toestanden kunnen voorkomen bij bewerkingen op een (verondersteld) open bestand :

notOpen

Een bewerking wordt uitgevoerd op een bestand dat niet (meer) geopend is.

endError

Een leesbewerking wordt uitgevoerd nadat de toestand van het bestand reeds is gewijzigd in EOL, EOF of EOT.

outsideFile

Het bestand wordt geadresseerd voor het begin of na het einde van het bestand.

Alle andere mogelijke fouten worden niet in de standaard-bibliotheek gedefinieerd. Voor deze fouten is de toestand otherError.

16.6 Het openen van een bestand

Een bestaand bestand wordt geopend met de procedure Open, een nieuw bestand met de procedure Create.

```
PROCEDURE Open    (VAR file      : File;
                       naam      : ARRAY OF CHAR;
                       binTekst  : BinTextMode;
                       writeMode : ReadWriteMode;
                   VAR status    : FileState);

PROCEDURE Create  (VAR file      : File;
                       naam      : ARRAY OF CHAR;
                       binTekst  : BinTextMode;
                       vervMode  : ReplaceMode;
                   VAR status    : FileState);
```

Deze beide procedures verbinden een bestandsvariabele <file> met een extern bestand met de bestandsnaam <naam>. Voor beide procedures wordt de toegangswijze tot het bestand vastgelegd : binair of tekst. Bij het aanroepen van Open specificeren we ook of we het bestand willen lezen en/of schrijven. Bij de procedure Create wordt altijd in een bestand geschreven. Hier geven we wel met <vervMode> aan hoe een bestaand bestand moet worden behandeld.

Voorbeelden :

```
FROM Files IMPORT File, BinTextMode, ReadWriteMode, FileState,
                  ReplaceMode, Open, Create;

VAR f, g   : File;
    status : FileState;

BEGIN
Open(f, "INVOER.DAT", textMode, readOnly, status);
...
Create(g, "PERSOON.IDX", binMode, replace, status);
...
END;
```

Het bestand f wordt geopend als een tekstbestand. Van dit bestand kan alleen worden gelezen. De externe naam is "INVOER.DAT". Het bestand g wordt gecreëerd als een binair bestand. Als dit bestand reeds bestaat, wordt het bestaande bestand overschreven.

Mogelijke toestanden na het uitvoeren van Open en Create :

```
Open   : ok, nameError, noFile, deviceError, otherError.
Create : ok, nameError, existingFile, deviceError, noMoreRoom,
         otherError.
```

16.7 Het afsluiten van een bestand

Als we een bestand in een programma niet meer nodig hebben, sluiten we het af met de bewerking Close of Remove :

```
PROCEDURE Close    (VAR file      : File;
                    VAR status    : FileState);

PROCEDURE Remove   (VAR file      : File;
                    VAR status    : FileState);
```

De procedures Close en Remove sluiten het bestand <file> af. Na het afsluiten is de bestandsvariabele in het programma niet meer beschikbaar. Met de procedure Close worden eerst alle gegevens, die eventueel nog in een interne buffer worden bewaard, naar het externe medium geschreven. Met de procedure Remove wordt het bestand van het externe medium gewist. De waarde van <status> geeft het resultaat weer van de afsluitbewerking : ok, device-Error, notOpen of otherError. Als tijdens de afsluitbewerking een fout voorkomt, wordt het bestand toch afgesloten.

16.8 Het instellen van een bestand

Door de bewerking Reset, Rewrite of Truncate wordt de positie ingesteld waar de volgende lees- of schrijfopdracht met het bestand wordt uitgevoerd :

```
PROCEDURE Reset   (    file      : File;
                   VAR status    : FileState);

PROCEDURE Rewrite (    file      : File;
                   VAR status    : FileState);

PROCEDURE Truncate(    file      : File;
                   VAR status    : FileState);
```

Door de procedure Reset wordt de leespositie ingesteld aan het begin van het bestand, zodat het bestand opnieuw kan worden gelezen. De procedure Rewrite stelt ook de positie in aan het begin van het bestand. Het bestand wordt logisch gewist en kan daarna opnieuw worden geschreven. De procedure Truncate wist alle gegevens na de huidige positie. Met de procedure Rewrite en Truncate moet het bestand met de eigenschap readWrite worden geopend.
De toestandswaarden van het bestand na deze bewerkingen zijn :

```
Reset    : ok, deviceError, accessError, notOpen, otherError;
Rewrite  : ok, deviceError, noMoreRoom, accessError, notOpen;
Truncate : ok, deviceError, accessError, notOpen, otherError;
```

16.9 De toestand en de naam van een bestand

De toestand van een bestand kan worden nagegaan met de bewerkingen EOF of State. Een foutieve toestand wordt weggewerkt met de bewerkingen ResetState. De bewerking GetFileName levert de bestandsnaam voor een bestandsvariabele. Met de procedure Flush worden de interne buffers geleegd.

```
PROCEDURE EOF     (    file      : File) : BOOLEAN;

PROCEDURE State   (    file      : File) : FileState;

PROCEDURE ResetState(  file      : File;
                   VAR status    : FileState);

PROCEDURE GetFileName( file      : File;
                   VAR naam      : ARRAY OF CHAR;
                   VAR status    : FileState);
```

```
PROCEDURE Flush   (    file       : File;
                   VAR status     : FileState);
```

De waarde van de functieprocedure EOF wordt TRUE als een leesopdracht van een bestand niet is gelukt. Dit betekent dat de procedure EOF niet vooruitziet. De algemene structuur van een programma voor de verwerking van een bestand is dan :

- met de REPEAT-opdracht :

```
(* het bestand 'in' is geopend voor leesopdrachten *)
REPEAT
  ReadString(in, string);
  IF NOT EOF(in)
  THEN
    (* verwerk de invoer *)
  END
UNTIL EOF(in);
```

- met de LOOP-opdracht :

```
(* het bestand 'in' is geopend voor leesopdrachten *)
LOOP
  ReadString(in, string);
  IF EOF(in)
  THEN
    EXIT
  END;
  (* verwerk de invoer *)
END;
```

- met de WHILE-opdracht :

```
(* het bestand 'in' is geopend voor leesopdrachten *)
ReadString(in, string);
WHILE NOT EOF(in) DO
  (* verwerk de invoer *)
  ReadString(in, string)
END;
```

De waarde van EOF wordt ook TRUE als een of andere in- of uitvoeropdracht mislukt. EOF heeft dus twee betekenissen : 'einde bestand' en 'fout'. Om dubbelzinnigheid in een programma te vermijden controleren we de toestand van het bestand. De waarde TRUE voor EOF en een toestand 'ok' betekent dat het einde van het bestand is bereikt. De waarde TRUE voor EOF en een toestand verschillend van 'ok' impliceert het voorkomen van een fout.

Het resultaat van een bewerking wordt gegeven door de parameter <status>, de toestand van een bestand door de functieprocedure State. Als we een bestand met een foutieve toestand verder verwerken wordt het programma gestopt.

De procedure Flush

Als we met bestanden werken, worden de in- en uitvoerbewerkingen niet steeds direct op een extern medium uitgevoerd. De gegevens worden meestal tijdelijk bewaard in een interne geheugenbuffer. Het besturingssysteem zorgt voor het vullen of leegmaken van deze buffers. Als de buffer leeg is bij een leesopdracht of als de buffer vol loopt bij een schrijfopdracht, grijpt het besturingssysteem in. Pas op dit ogenblik worden bepaalde fouten, bijvoorbeeld noMoreRoom, deviceError, ontdekt.

Door de procedure Flush worden de interne buffers naar het externe medium geschreven. Zo voorkomen we het verlies van gegevens in de interne buffers ten gevolge van eventuele storingen en worden sommige in- of uitvoerfouten direct vastgesteld. De Flush-bewerking is daarentegen weinig efficiënt.

De procedure GetFileName

Met de procedure GetFileName bepalen we de volledige en ondubbelzinnige naam van een bestandsvariabele.

De toestandswaarden na het uitvoeren van de procedures ResetState, Flush en GetFileName kunnen zijn :

```
ResetState  : ok, otherError;
Flush       : ok, deviceError, notOpen, otherError;
GetFileName : ok, deviceError, notOpen, otherError.
```

16.10 De module Binary

De module Binary wordt steeds samen met de module Files gebruikt. De module Files gebruiken we voor het openen en afsluiten van een bestand en voor het onderzoeken van de toestand van een bestand. De module Binary gebruiken we voor het uitvoeren van lees- en schrijfopdrachten. De gegevens worden zonder conversie tussen het interne geheugen en het externe medium getransporteerd. Deze vorm van in- en uitvoer is zeer efficiënt omdat conversies nogal wat rekentijd vergen. Een binair bestand kan niet worden afgedrukt of direct door gebruikers worden gelezen.

```
DEFINITION MODULE Binary;
FROM Files  IMPORT File, FileState;
FROM SYSTEM IMPORT BYTECOUNT;
FROM SYSTEM IMPORT BYTE, WORD, ADDRESS;
EXPORT QUALIFIED
  (* proc *) ReadByte,  ReadWord,  ReadBlock,  ReadBytes,
             WriteByte, WriteWord, WriteBlock, WriteBytes;


PROCEDURE ReadByte  (    file    : File;
                     VAR byte    : BYTE;
                     VAR status  : FileState);

PROCEDURE ReadWord  (    file    : File;
                     VAR word    : WORD;
                     VAR status  : FileState);

PROCEDURE ReadBlock (    file    : File;
                     VAR block   : ARRAY OF BYTE;
                     VAR status  : FileState);

PROCEDURE ReadBytes (    file    : File;
                         adres   : ADDRESS;
                         bytes   : BYTECOUNT;
                     VAR gelezen : BYTECOUNT;
                     VAR status  : FileState);


PROCEDURE WriteByte (    file    : File;
                         byte    : BYTE;
                     VAR status  : FileState);

PROCEDURE WriteWord (    file    : File;
                         word    : WORD;
                     VAR status  : FileState);

PROCEDURE WriteBlock(    file    : File;
                         block   : ARRAY OF BYTE;
                     VAR status  : FileState);

PROCEDURE WriteBytes(    file    : File;
                         adres   : ADDRESS;
                         bytes   : BYTECOUNT;
                     VAR status  : FileState);
END Binary.
```

Met de procedure ReadByte en WriteByte kan één enkele byte van het bestand worden gelezen of naar het bestand worden geschreven. De procedure ReadWord en WriteWord hebben hetzelfde effect, maar dan voor één woord. Als de laatste byte is gelezen door ReadWord, maar minder is dan een woord, is de waarde van <status> gelijk aan 'endError'.

Met de procedures ReadBlock en WriteBlock wordt een variabele met een willekeurig type en met een willekeurige lengte geschreven naar of gelezen van een bestand.

Met de procedures ReadBytes en WriteBytes worden <bytes> bytes gelezen van of geschreven naar een bestand. Het adres van de interne buffer wordt gegeven met de actuele parameter <adres>. Bij de procedure ReadBytes wordt het aantal te lezen bytes gegeven door <bytes>. Het werkelijke aantal gelezen bytes wordt gegeven door <gelezen>. Deze waarden kunnen van elkaar verschillen aan het einde van het bestand (EOF is TRUE en status is 'ok') of bij het optreden van een fout (EOF is TRUE en status verschilt van 'ok').

16.11 De module FilePositions

Met de module FilePositions kunnen we de lees/schrijfpositie in een bestand opvragen, berekenen en instellen. Deze module wordt meestal samen met de module Binary gebruikt voor het verwerken van direct adresseerbare bestanden. De module FilePositions maakt het mogelijk gegevens te lezen van of te schrijven naar een willekeurige plaats in het bestand. Als we schrijven voor het einde van het bestand, worden de aanwezige gegevens overschreven. Als we schrijven aan het einde van het bestand, wordt dit uitgebreid.

```
DEFINITION MODULE FilePositions;
FROM Files   IMPORT File, FileState;
FROM SYSTEM  IMPORT ADDRESSINC;
EXPORT QUALIFIED
  (* type *) FilePosition,
  (* proc *) GetFilePos, SetFilePos,
             CalcFilePos, GetEOF, GetBOF;

TYPE FilePosition = RECORD
       (* definitie is systeemafhankelijk *)
     END;
```

```
PROCEDURE GetFilePos  (    file   : File;
                       VAR pos    : FilePosition);

PROCEDURE GetEOF      (    file   : File;
                       VAR pos    : FilePosition);

PROCEDURE GetBOF      (    file   : File;
                       VAR pos    : FilePosition);

PROCEDURE CalcFilePos (    file   : File;
                       VAR pos    : FilePosition;
                           aantal : INTEGER;
                           lengte : ADDRESSINC);

PROCEDURE SetFilePos  (    file   : File;
                           pos    : FilePosition;
                       VAR status : FileState);
END FilePositions.
```

Het type FilePosition wordt geïmplementeerd als een RECORD, zodat een waarde met type FilePosition eenvoudig naar een bestand kan worden geschreven of van een bestand kan worden gelezen. Met behulp van de waarden voor FilePosition voor de verschillende records in een gegevensbestand, kunnen indexen worden opgebouwd. Met de modulen Binary en FilePositions beschikken we over de nodige procedures voor het construeren van bestanden met index-sequentiële toegang.

De definitie van het type FilePosition is afhankelijk van het systeem waarop de Modula-2 programma's draaien. We definiëren het type FilePosition hier als :

```
TYPE FilePosition = RECORD
       laag  : CARDINAL;
       hoog  : CARDINAL
     END;
```

De positie in een bestand wordt dan gegeven door de uitdrukking :

```
          positie = hoog * (MAX(CARDINAL) + 1) + laag
```

```
PROCEDURE GetFilePos  (    file   : File;
                       VAR pos    : FilePosition);

PROCEDURE GetEOF      (    file   : File;
                       VAR pos    : FilePosition);

PROCEDURE GetBOF      (    file   : File;
                       VAR pos    : FilePosition);
```

Met de procedures GetFilePos, GetBOF en GetEOF bepalen we respectievelijk de huidige positie in het bestand, de positie van het begin en de positie van het einde van het bestand.

```
PROCEDURE CalcFilePos (    file   : File;
                       VAR pos    : FilePosition;
                           aantal : INTEGER;
                           lengte : ADDRESSINC);

PROCEDURE SetFilePos  (    file   : File;
                           pos    : FilePosition;
                       VAR status : FileState);
```

Met de procedure CalcFilePos berekenen we de positie <pos> in het bestand <file>. De nieuwe positie berekenen we ten opzichte van de huidige positie, rekening houdend met de lengte <lengte> van een gegevenselement en het aantal <aantal> elementen. Dit aantal kan ook negatief zijn : in dit geval verplaatsen we de lees/schrijfpositie naar het begin van het bestand. De positiebepaling kan ook geschieden ten opzichte van het begin of het einde van het bestand. Hiervoor gebruiken we dan de procedures GetBOF of GetEOF.

Na het bepalen van de positie stellen we de plaats in een bestand in met de procedure SetFilePos. Als we een plaats voor BOF of na EOF willen instellen, wordt de huidige plaats niet gewijzigd en is de waarde van <status> gelijk aan 'outsideFile'.

Voorbeeld : het kopiëren van een binair bestand

```
MODULE Kopieer;
FROM Terminal   IMPORT ReadString,
                       WriteString, WriteLn;
FROM Files      IMPORT File, FileState,
                       BinTextMode, ReadWriteMode, ReplaceMode,
                       Create, Open, Close;
FROM Binary     IMPORT ReadBytes, WriteBytes;
FROM SYSTEM     IMPORT ADR;
```

```
PROCEDURE VerwerkStatus(status : FileState);
BEGIN
CASE status OF
  nameError    :
  WriteString("syntaxisfout in de bestandsnaam")
| noFile       :
  WriteString("bestand is niet aanwezig")
| existingFile :
  WriteString("bestand bestaat reeds")
| deviceError  :
  WriteString("fout in de in- of uitvoerapparatuur")
| noMoreRoom   :
  WriteString("onvoldoende opslagruimte")
| accessError  :
  WriteString("foutieve toegang tot het bestand")
| notOpen      :
  WriteString("bestand is niet geopend")
| endError     :
  WriteString("leesopdracht na einde bestand")
| outsideFile  :
  WriteString("positie voor begin of na einde bestand")
ELSE
  WriteString("fout niet gedefinieerd")
END;
WriteLn;
HALT
END VerwerkStatus;

(* definitie geheugenbuffer *)
CONST bufferLengte = 1ØØØH;
VAR   buffer       : ARRAY [1..bufferLengte] OF CHAR;
      gelezen      : [Ø..bufferLengte];

(* definitie van de bestanden *)
CONST naamLengte   = 13;

TYPE  Bestandsnaam = ARRAY [Ø..naamLengte - 1] OF CHAR;
VAR   f, g         : File;
      naam         : Bestandsnaam;
      status       : FileState;

BEGIN
(* initieer het bronbestand *)
WriteString("naam bronbestand : ");
ReadString(naam);
Open(f, naam, binMode, readOnly, status);
IF status # ok
THEN
  VerwerkStatus(status)
END;
```

```
 (* initieer het doelbestand *)
WriteString("naam doelbestand : ");
ReadString(naam);
Create(g, naam, binMode, noReplace, status);
IF status # ok
THEN
  VerwerkStatus(status)
END;

(* kopieer het bronbestand naar het doelbestand *)
REPEAT
  ReadBytes(f, ADR(buffer), bufferLengte, gelezen, status);
  IF status # ok
  THEN
    VerwerkStatus(status)
  END;
  WriteBytes(g, ADR(buffer), gelezen, status);
  IF status # ok
  THEN
    VerwerkStatus(status)
  END
UNTIL gelezen < bufferLengte;

(* sluit de bestanden af *)
Close(f, status);
Close(g, status);
IF status # ok
THEN
  VerwerkStatus(status)
END

END Kopieer.
```

17 Tekstbestanden

17.1 De module Text

De module Text wordt steeds samen met de module Files gebruikt. Met de module Files wordt een bestand geopend of afgesloten. De module Text levert de bewerkingen om uit een bestand te lezen of naar een bestand te schrijven. Met de lees- en schrijfbewerkingen van de module Text worden de gegevens in tekstvorm bewaard. We noemen dit een tekstbestand. Een tekstbestand kan direct worden afgedrukt of zonder verdere verwerking door gebuikers worden gelezen. De bewerkingen met een tekstbestand zijn over het algemeen minder efficiënt dan die met een binair bestand.

```
DEFINITION MODULE Text;
FROM Files IMPORT File, FileState;

EXPORT QUALIFIED
  (* proc *) EOL,
             ReadChar, ReadString, ReadLn,
             UndoRead, CondRead,
             WriteChar, WriteString, WriteLn;

PROCEDURE EOL         (    file   : File) : BOOLEAN;

PROCEDURE ReadChar    (    file   : File;
                       VAR teken  : CHAR;
                       VAR status : FileState);

PROCEDURE ReadString  (    file   : File;
                       VAR str    : ARRAY OF CHAR;
                       VAR status : FileState);

PROCEDURE ReadLn      (    file   : File;
                       VAR status : FileState);
```

```
PROCEDURE UndoRead    (    file   : File;
                       VAR status : FileState);

PROCEDURE CondRead    (    file   : File;
                       VAR teken  : CHAR;
                       VAR succes : BOOLEAN;
                       VAR status : FileState);

PROCEDURE WriteChar   (    file   : File;
                           teken  : CHAR;
                       VAR status : FileState);

PROCEDURE WriteString (    file   : File;
                           str    : ARRAY OF CHAR;
                       VAR status : FileState);

PROCEDURE WriteLn     (    file   : File;

                       VAR status : FileState);

END Text.
```

17.2 De leesopdrachten

```
PROCEDURE EOL         (    file   : File) : BOOLEAN;
```

De waarde van de functieprocedure EOL wordt TRUE als een leesopdracht mislukt omdat de regel ten einde is. De toestand 'einde regel' kan slechts worden opgeheven met de procedure ReadLn. Vòòr het einde van de regel is de waarde FALSE.

```
PROCEDURE ReadChar    (    file   : File;
                       VAR teken  : CHAR;
                       VAR status : FileState);
```

De procedure ReadChar leest een teken <teken> uit het bestand <file>. De procedure ReadChar leest uitsluitend tekens en kan dus de toestand 'einde regel' niet wijzigen.

```
PROCEDURE ReadString  (    file   : File;
                       VAR str    : ARRAY OF CHAR;
                       VAR status : FileState);
```

De procedure ReadString leest een string <str> uit het bestand <file>. De leesopdracht eindigt bij een van de volgende condities :

- de toestand 'einde regel' of 'einde bestand' wordt TRUE. Als de stringvariabele nog niet volledig is gevuld, worden de geldige tekens van de string afgesloten met het teken endStr;
- de stringvariabele is volledig gevuld.

De toestand 'einde regel' kan slechts worden opgeheven met de procedure ReadLn. We kunnen een tekstregel samenstellen uit verschillende velden, met één stringwaarde per veld. Met de procedure ReadString kan elk veld afzonderlijk worden gelezen.

```
PROCEDURE ReadLn       (    file   : File;
                        VAR status : FileState);
```

De procedure ReadLn leest alle tekens van een regel tot en met het besturingsteken 'einde regel'. Met de procedure ReadLn wordt de toestand 'einde regel' opgeheven.

```
PROCEDURE UndoRead     (    file   : File;
                        VAR status : FileState);
```

De procedure UndoRead plaatst het laatst gelezen teken van het bestand weer terug in dit bestand, zodat het opnieuw kan worden gelezen. Bij de aanroep van de procedure UndoRead hoeft dit teken evenwel niet te worden gespecificeerd : de implementatiemodule houdt hiervoor zelf de nodige gegevens bij. Bij de aanroep van UndoRead en de toestand 'einde regel' of 'einde bestand', wordt de desbetreffende toestand opgeheven.

```
PROCEDURE CondRead     (    file   : File;
                        VAR teken  : CHAR;
                        VAR succes : BOOLEAN;
                        VAR status : FileState);
```

De leesopdrachten ReadLn, ReadString en ReadChar wachten op invoer. De procedure CondRead wacht echter niet op de invoer van een teken. De procedure CondRead tracht een teken te lezen uit het invoerbestand <file>. Als een teken aanwezig is, wordt de waarde van <succes> TRUE. Als er geen teken aanwezig is, wordt de waarde van <succes> FALSE en is de waarde van <teken> onbepaald. Als het bestand aanwezig is op een extern medium, is de waarde van <succes> steeds TRUE zolang de waarde van EOF FALSE is. Bij een interactief bestand is de waarde van <succes> slechts TRUE als het teken door de gebruiker zelf wordt ingevoerd.

17.3 De schrijfopdrachten

```
PROCEDURE WriteChar   (    file   : File;
                           teken  : CHAR;
                       VAR status : FileState);

PROCEDURE WriteString (    file   : File;
                           str    : ARRAY OF CHAR;
                       VAR status : FileState);
```

De procedures WriteChar en WriteString schrijven respectievelijk een teken of een string naar het bestand <file>. Als het teken endStr in een stringwaarde voorkomt voor de gedeclareerde lengte van de string, wordt dit door de procedure WriteString beschouwd als het einde van de string.

```
PROCEDURE WriteLn     (    file   : File;
                       VAR status : FileState);
```

De procedure WriteLn schrijft het teken 'einde regel' naar het bestand <file>.

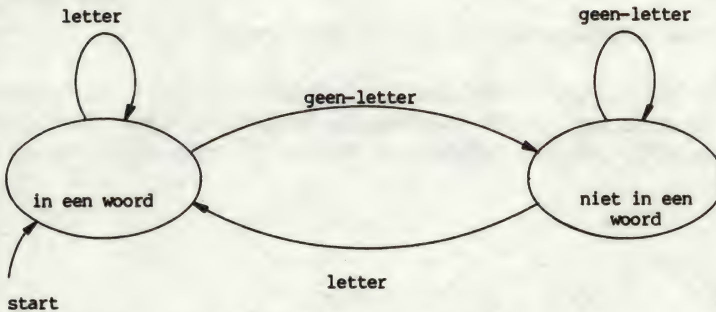
Voorbeeld : woordenlijst

Van een gegeven tekst willen we een lijst maken met alle woorden die in de tekst voorkomen. De tekst is opgeslagen op een extern bestand. Hij bestaat uit een aantal regels, elke regel bevat een of meer woorden. We definiëren een woord als een reeks letters die wordt omsloten door een geen-letter of een besturingsteken.

Voor de oplossing van het probleem formuleren we een algoritme waarbij we rekening houden met de toestand waarin het programma zich op een bepaald ogenblik bevindt. Voor het programma beschouwen we de twee toestanden 'in een woord' en 'niet in een woord'. Als we een letter van het invoerbestand lezen en de toestand is 'in een woord', dan wordt de letter toegevoegd aan het einde van het woorddeel. Bij de toestand 'niet in een woord' geeft een letter het begin aan van een nieuw woord. Dit betekent dat de toestand moet wijzigen van 'niet in een woord' naar 'in een woord'.

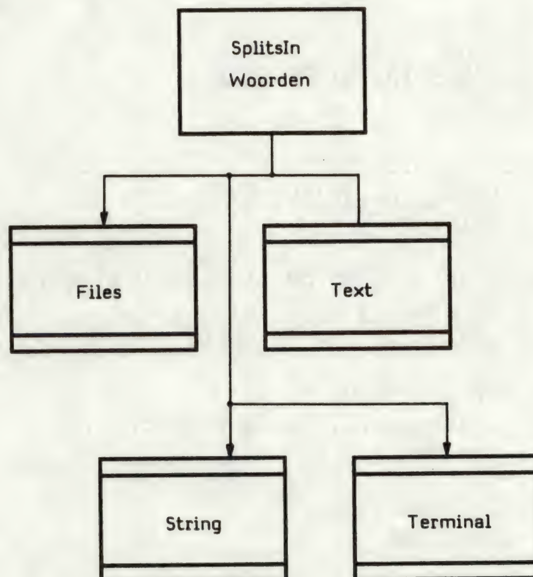
Als een geen-letter van het invoerbestand wordt gelezen en de toestand is 'in een woord', betekent dit het einde van het huidige woord. Dit woord kan naar het uitvoerbestand worden geschreven. De toestand 'niet in een woord' blijft behouden bij het lezen van een geen-letter.

We stellen de werking van het programma voor met een status-overgangsdiagram :



figuur 17.1 Statusovergangsdiagram

Voor de implementatie van deze oplossing beschouwen we de in- en uitvoerbestanden als tekstbestanden en gebruiken we bewerkingen van de module Text. De bestanden worden geopend en afgesloten met de bewerkingen Files.Open en Files.Close. De bestandsnamen worden interactief ingevoerd via het werkstation. Hiervoor gebruiken we de module Terminal. Voor de opbouw van een woord gebruiken we de operatie String.Concat. Dit geeft het volgende abstractieschema :



figuur 17.2 Abstractieschema 'Splits in woorden'

Voor de toestand waarin het programma zich bevindt definiëren we het type Toestand :

```
TYPE Toestand = (inEenWoord, nietInEenWoord);
```

Voor de bestanden gebruiken we variabelen 'invoer' en 'uitvoer'. Voor het afhandelen van de eventuele fouten tijdens de verwerking van een bestand, kan de procedure VerwerkStatus opnieuw worden gebruikt.

```
MODULE SplitsInWoorden;
FROM Files      IMPORT File, FileState,
                       BinTextMode, ReadWriteMode, ReplaceMode,
                       Open, Create, Close, EOF;
FROM Text       IMPORT EOL, ReadChar, ReadLn,
                       WriteString, WriteLn;
FROM String     IMPORT Concat, Assign;
IMPORT Terminal;

PROCEDURE VerwerkStatus(status : FileState);
BEGIN
...
END VerwerkStatus;

(* variabelen voor de invoer *)

CONST eol       = 15C;
VAR  teken      : CHAR;
     tekenStr   : ARRAY [0..0] OF CHAR;
     invoer     : File;

(* variabelen voor de uitvoer *)
     woord      : ARRAY [0..20] OF CHAR;
     uitvoer    : File;

(* variabelen voor de naam en de toestand van een bestand *)
     status     : FileState;
     naam       : ARRAY [0..13] OF CHAR;

(* toestand van het programma *)
TYPE Toestand  = (inEenWoord, nietInEenWoord);
VAR  toestand  : Toestand;

(* resultaat van een bewerking *)
VAR  succes     : BOOLEAN;
```

```
BEGIN
(* open het invoerbestand *)
Terminal.WriteString('invoerbestand : ');
Terminal.ReadString(naam);
Terminal.WriteLn;
Open(invoer, naam, textMode, readOnly, status);
IF status # ok
THEN
  VerwerkStatus(status)
END;

(* open het uitvoerbestand *)
Terminal.WriteString('uitvoerbestand : ');
Terminal.ReadString(naam);
Terminal.WriteLn;
Create(uitvoer, naam, textMode, noReplace, status);
IF status # ok
THEN
  VerwerkStatus(status)
END;

(* initieer de toestand en het huidige woord *)
toestand := nietInEenWoord;
woord := '';

(* verwerk het invoerbestand *)

(* lees het eerste teken *)
ReadChar(invoer, teken, status);

(* verwerk het teken *)
WHILE NOT EOF(invoer) DO

  (* verwerk de toestand 'einde regel' *)
  IF EOL(invoer)
  THEN
    ReadLn(invoer, status);
    teken := eol
  END;

  (* zet een hoofdletter om tot een kleine letter *)
  IF ('A' <= teken) AND (teken <= 'Z')
  THEN
    teken := CHR(ORD(teken) - ORD('A') + ORD('a'))
  END;
```

```
  (* verwerk het teken *)
  CASE toestand OF
    inEenWoord :
    IF ('a' <= teken) AND (teken <= 'z')
    THEN
      tekenStr[Ø] := teken;
      Concat(woord, tekenStr, woord, succes)
    ELSE
      WriteString(uitvoer, woord, status);
      WriteLn(uitvoer, status);
      IF status # ok
      THEN
        VerwerkStatus(status)
      END;
      toestand := nietInEenWoord
    END

  | nietInEenWoord :
    IF ('a' <= teken) AND (teken <= 'z')
    THEN
      tekenStr[Ø] := teken;
      Assign(tekenStr, woord, succes);
      toestand := inEenWoord
    END
  END;

  ReadChar(invoer, teken, status)
END;

(* sluit de bestanden af *)
Close(invoer, status);
Close(uitvoer, status);
IF status # ok
THEN
  VerwerkStatus(status)
END

END SplitsInWoorden.
```

17.4 De module NumberIO

Met de module NumberIO kunnen we getallen met type INTEGER of met type CARDINAL lezen van of schrijven naar een tekstbestand. Natuurlijke getallen kunnen ook met een willekeurig grondtal worden gelezen of geschreven.

```
DEFINITION MODULE NumberIO;
FROM Files  IMPORT File, FileState;
FROM SYSTEM IMPORT WORD;

EXPORT QUALIFIED
  (* proc *) ReadInt, ReadCard, ReadNum,
             WriteInt, WriteCard, WriteNum;

PROCEDURE ReadInt  (    file     : File;
                    VAR integer  : INTEGER;
                    VAR succes   : BOOLEAN;
                    VAR status   : FileState);

PROCEDURE ReadCard (    file     : File;
                    VAR cardinal : CARDINAL;
                    VAR succes   : BOOLEAN;
                    VAR status   : FileState);

PROCEDURE ReadNum  (    file     : File;
                    VAR getal    : WORD;
                        basis    : CARDINAL;
                    VAR succes   : BOOLEAN;
                    VAR status   : FileState);

PROCEDURE WriteInt (    file     : File;
                        integer  : INTEGER;
                        breedte  : CARDINAL;
                    VAR status   : FileState);

PROCEDURE WriteCard(    file     : File;
                        cardinal : CARDINAL;
                        breedte  : CARDINAL;
                    VAR status   : FileState);

PROCEDURE WriteNum (    file     : File;
                        getal    : WORD;
                        basis    : CARDINAL;
                        breedte  : CARDINAL;
                    VAR status   : FileState);

END NumberIO.
```

Met de procedures ReadInt en ReadCard kunnen we de inhoud van een tekstbestand interpreteren als getallen met type INTEGER of CARDINAL. Door deze procedures worden getallen van een tekstbestand gelezen. De leesprocedures werken als volgt : eerst worden de voorloopspaties gelezen. Met deze spaties wordt geen rekening gehouden. Het teken en de cijfers worden omgezet in de inwendige

representatie van het getal. De leesopdracht eindigt bij het eerste teken dat geen cijfer is. De waarde van <succes> is FALSE als het eerste teken geen cijfer of een '+'- of '-'-teken is of als de getalwaarde overloopt.

Met de procedures WriteInt en WriteCard worden getallen naar een tekstbestand beschreven. Bij deze schrijfopdrachten worden ten minste <breedte> tekens naar het bestand geschreven. Als deze <breedte> te klein is, worden automatisch de nodige tekens toegevoegd.

Met de procedures ReadNum en WriteNum worden getallen met het grondtal twee tot eenendertig van een bestand gelezen of naar een bestand geschreven.

Oefeningen

1. Schrijf een programmamodule MengWoordenlijst voor het mengen van twee geordende woordenlijsten. Stel met dit programma een woordenboek samen.

2. Ontwerp en implementeer een eenvoudige procedure voor spellingscontrole. Dit programma kan bestaan uit de volgende stappen :
- splits in woorden;
- orden de woordenlijst;
- wis de herhalingen;
- vergelijk met het woordenboekbestand.

3. Veel documenten, die met een computersysteem worden aangemaakt, bevatten bijzondere opdrachten voor de vormgeving. Deze opdrachten worden gebruikt voor het weergeven van de tekst met een randapparaat, bijvoorbeeld een grafisch scherm of een drukker. Veronderstel de volgende syntaxis voor deze opdrachten :
- elke vormgevingsopdracht begint op een nieuwe regel;
- de syntaxis voor de opdracht is :

'.' letter (letter | cijfer) [parameter]

Beschouw bijvoorbeeld enkele opdrachten van het UNIX-ROFF-programma :

opdracht	betekenis
.ad	activeer rechts uitlijnen;
.ar	Arabische bladzijdenummering;
.br	begin een nieuwe regel;
.bl n	voeg n lege regels toe;

Wijzig de procedure Woordenlijst zodat deze opdrachten niet in de woordenlijst worden opgenomen.

4. Ontwerp een verzameling opdrachten voor de vormgeving van een tekst. Definieer deze opdrachten in een definitiemodule.

5. Ontwerp een programma voor het interpreteren van een tekst met vormgevingsopdrachten. Gebruik de module Beeldscherm voor de weergave van de tekst op het werkstation.

6. Generaliseer de oplossing van opgave 5 zodat een willekeurig uitvoerapparaat kan worden aangestuurd, bijvoorbeeld het werkstation, een kettingdrukker, een laserdrukker.

18 De standaardin- en -uitvoer

18.1 De module SimpleIO

De module SimpleIO is bedoeld voor een eenvoudige interactie tussen de gebruiker en het systeem via de standaardin- en -uitvoer. Met de bewerkingen van deze module kunnen we de eventuele fouten echter niet beheren. Het aanroepen van de procedures is vrij eenvoudig door de aanwezigheid van impliciete bestandsparameters.

```
MODULE SimpleIO;
FROM SYSTEM IMPORT WORD;
EXPORT QUALIFIED
  (* proc *) EOT, EOL,
             ReadChar, ReadString, ReadLn,
             ReadInt, ReadCard, ReadNum,
             CondRead, UndoRead,
             WriteChar, WriteString, WriteLn,
             WriteInt, WriteCard, WriteNum;

PROCEDURE EOT        () : BOOLEAN;
PROCEDURE EOL        () : BOOLEAN;

PROCEDURE ReadChar   (VAR teken    : CHAR);
PROCEDURE ReadString (VAR str      : ARRAY OF CHAR);
PROCEDURE ReadLn;

PROCEDURE ReadInt    (VAR integer  : INTEGER;
                      VAR succes   : BOOLEAN);
PROCEDURE ReadCard   (VAR cardinal : CARDINAL;
                      VAR succes   : BOOLEAN);
PROCEDURE ReadNum    (VAR getal    : WORD;
                          basis    : CARDINAL;
                      VAR succes   : BOOLEAN);

PROCEDURE CondRead   (VAR teken    : CHAR;
                      VAR succes   : BOOLEAN);
PROCEDURE UndoRead   ();
```

```
PROCEDURE WriteChar  (    teken    : CHAR);
PROCEDURE WriteString(    str      : ARRAY OF CHAR);
PROCEDURE WriteLn;
PROCEDURE WriteInt   (    integer  : INTEGER;
                          breedte  : CARDINAL);
PROCEDURE WriteCard  (    cardinal : CARDINAL;
                          breedte  : CARDINAL);
PROCEDURE WriteNum   (    getal    : WORD;
                          basis    : CARDINAL;
                          breedte  : CARDINAL);

END SimpleIO.
```

De procedures EOL, ReadChar, ReadString, ReadLn, UndoRead, CondRead, ReadInt, ReadCard en ReadNum hebben dezelfde betekenis als de overeenkomstige procedures in de modulen Text en NumberIO. De bestandsparameter <file> wordt hier niet vermeld. De functieprocedure EOT heeft de betekenis van Files.EOF. Ook de procedures WriteChar, WriteString, WriteLn, WriteInt, WriteCard en WriteNum hebben dezelfde betekenis als in de modulen Text en NumberIO.

18.2 De module RealIO

De module RealIO voegt aan de module SimpleIO de bewerkingen voor het lezen en schrijven van getallen met type REAL toe.

```
DEFINITION MODULE RealIO;
EXPORT QUALIFIED
  (* proc *) ReadReal, WriteReal;

PROCEDURE ReadReal (VAR real      : REAL;
                    VAR succes    : BOOLEAN);

PROCEDURE WriteReal(    real      : REAL;
                        breedte   : CARDINAL;
                        decPlaats : CARDINAL);
END RealIO.
```

Met de procedure WriteReal specificeren we met <breedte> hoeveel tekens ten minste moeten worden afgedrukt. Als dit aantal te klein is om het getal weer te geven, wordt de wetenschappelijke notatie gebruikt. Als ook hiervoor te weinig ruimte is, worden de nodige tekens toegevoegd en wordt het getal weergegeven in de

gespecificeerde vorm. Het aantal tekens na de decimale punt specificeren we met <decPlaats>. Een negatieve waarde voor deze parameter impliceert de wetenschappelijke notatie. Een teken wordt alleen weergegeven voor negatieve waarden.

18.3 De module StandardIO

De modulen SimpleIO en RealIO werken systeemgekozen met de standaardinvoer, het toetsenbord, en de standaarduitvoer, het beeldscherm. De module StandardIO beschikt over de bewerkingen om de standaardin- en -uitvoer ook aan een extern bestand te koppelen. De automatische weergave van de standaardinvoer naar de standaarduitvoer, de echo, en de aanmaak van een logbestand worden eveneens ingesteld met procedures van deze module.

```
DEFINITION MODULE StandardIO;
FROM Files IMPORT File;
EXPORT QUALIFIED
  (* type *) EchoMode, Logmode,
  (* proc *) SetInput, GetInput, SetOutput, GetOutput,
             SetEchoMode, GetEchoMode,
             GetErrorOutput, GetErrorInput,
             SetLogMode, GetLogMode,
             SetLog, GetLog;
(* standaardin- en -uitvoer *)
PROCEDURE SetInput       (    input       : File);
PROCEDURE GetInput       (VAR input       : File);
PROCEDURE SetOutput      (    output      : File);
PROCEDURE GetOutput      (VAR output      : File);
(* besturing van de echo van standaardin- naar -uitvoer *)
TYPE EchoMode = (echo, noEcho);
PROCEDURE SetEchoMode    (    mode        : EchoMode);
PROCEDURE GetEchoMode    () : EchoMode;
(* foutenbestanden *)
PROCEDURE GetErrorOutput(VAR foutbestand : File);
PROCEDURE GetErrorInput (VAR foutbestand : File);
(* instelling van het logbestand *)
TYPE LogMode = (loggingOn, loggingOff);
PROCEDURE SetLog         (    log         : File);
PROCEDURE GetLog         (VAR log         : File);
PROCEDURE SetLogMode     (    mode        : LogMode);
PROCEDURE GetLogMode     () : LogMode;

END StandardIO.
```

De procedure SetInput verbindt de standaardinvoer met de bestandsvariabele <input>. Dit bestand wordt eerst geopend met de opdracht Files.Open. De procedure SetOutput verbindt de standaarduitvoer aan de bestandsvariabele <output>. Het bestand <output> wordt eerst geopend met de procedure File.Open of File.Create. Met de procedures SetInput en SetOutput kunnen we dus de standaardin- en -uitvoer aan een willekeurig bestand koppelen. We beschikken dan over gelijksoortige faciliteiten voor 'redirection' als die welke door besturingssystemen worden geboden.

Voorbeeld :

```
FROM Files      IMPORT Open, ReadWriteMode, binTextMode,
                       FileState;
FROM StandardIO IMPORT SetInput;
FROM SimpleIO   IMPORT ReadChar, ReadLn;

VAR f, g  : File;
    state : FileState;
```

Met de opdrachten

```
Open(f, 'ABC.XYZ', textMode, readOnly, status);
SetInput(f);
```

wordt het externe bestand 'ABC.XYZ' geopend als een tekstbestand. Daarna wordt dit bestand verbonden met de standaardinvoer. We kunnen dit bestand nu lezen met de bewerkingen van de modulen SimpleIO of RealIO, bijvoorbeeld

```
ReadChar(teken);
ReadLn.
```

De procedures GetInput en GetOutput leveren de waarde af van de bestandsvariabale die met de standaardin- of -uitvoer is verbonden.

Voorbeeld :

```
Open(f, 'ABC.XYZ', textMode, readOnly, status);
SetInput(f);
...
GetInput(g);
```

Met de procedure GetInput wordt aan de variabele g de waarde f toegekend. De bestandsvariabelen f en g verwijzen nu naar hetzelfde bestand.

Het automatisch kopiëren van de standaardinvoer naar de standaarduitvoer wordt ingesteld met de procedure SetEchoMode. De waarden voor deze instelling worden gegeven door het type EchoMode. Met de waarde 'echo' wordt de invoer naar de uitvoer gekopieerd. Voor de invoer zonder echo stellen we de waarde 'noEcho' in. We gebruiken deze instelling bijvoorbeeld bij de invoer van een wachtwoord. De functieprocedure GetEchoMode levert de ingestelde waarden af.

Alle uitvoer naar de standaarduitvoer kan ook naar een logbestand worden geschreven. De procedure SetLog verbindt de standaarduitvoer met de bestandsvariabele <log>. Het bestand <log> wordt eerst geopend met de procedure File.Open of File.Create. Het kopiëren naar het logbestand wordt ingesteld met de procedure SetLogMode. De waarden voor de instelling worden gegeven door het type LogMode. Met de waarde 'loggingOn' wordt de standaarduitvoer gekopieerd naar het logbestand. De standaarduitvoer verkrijgen we niet met 'loggingOff'. Dit is de systeemgekozen toestand. De functieprocedure GetLogMode levert de waarde af voor de huidige toestand betreffende het aanmaken van het logbestand.

Met de eigenschappen 'echo' en 'loggingOn' kunnen we aldus de standaarduitvoer naar het beeldscherm en ook naar een extern bestand schrijven. De inhoud van het externe bestand is dan als het ware een fotografische afdruk van het beeldscherm.

Voorbeeld :
We geven nog eens de programmamodule SplitsInWoorden maar nu met de bewerkingen voor de standaardin- en -uitvoer. In deze oplossing gebruiken we de module SimpleIO in plaats van de module Text. Ook worden bewerkingen ingevoerd van de module StandardIO voor de instelling van de echo en voor het verbinden van de externe bestanden met de standaardin- en -uitvoer. De lees- en schrijfopdrachten worden met de bewerkingen van de module SimpleIO vereenvoudigd tot

```
ReadChar(teken)
```

of

```
WriteString(woord)
```

in plaats van de opdrachten van de module Text

```
ReadChar(invoer, teken, status)
```

of

```
WriteString(uitvoer, teken, status)
```

Tijdens het verwerken van de bestanden kan de toestand echter niet worden gecontroleerd. Bij een eventuele fout wordt het programma onderbroken.

We verkrijgen aldus de volgende module :

```
MODULE SplitsInWoorden;
FROM Files      IMPORT File, FileState,
                       BinTextMode, ReadWriteMode, ReplaceMode,
                       Open, Create, Close;
FROM SimpleIO   IMPORT EOT, EOL, ReadChar, ReadLn,
                       WriteString, WriteLn;
FROM String     IMPORT Concat, Assign;
FROM StandardIO IMPORT SetInput, SetOutput, SetEchoMode,
                       EchoMode;
IMPORT Terminal;


PROCEDURE VerwerkStatus(status : FileState);
BEGIN
...
END VerwerkStatus;


(* variabelen voor de invoer *)
CONST eol       : 15C;
VAR  teken      : CHAR;
     tekenStr   : ARRAY [0..0] OF CHAR;
     invoer     : File;

(* variabelen voor de uitvoer *)
     woord      : ARRAY [0..20] OF CHAR;
     uitvoer    : File;

(* variabelen voor de naam en de toestand van een bestand *)
     status     : FileState;
     naam       : ARRAY [0..13] OF CHAR;

(* toestand van het programma *)
TYPE Toestand  = (inEenWoord, nietInEenWoord);
VAR  toestand  : Toestand;

(* resultaat van een bewerking *)
VAR  succes     : BOOLEAN;


BEGIN
(* afzetten van de echo *)
SetEchoMode(noEcho);
```

```
(* open het invoerbestand *)
Terminal.WriteString('invoerbestand : ');
Terminal.ReadString(naam);
Terminal.WriteLn;
Open(invoer, naam, textMode, readOnly, status);
IF status # ok
THEN
  VerwerkStatus(status)
END;
SetInput(invoer);

(* open het uitvoerbestand *)
Terminal.WriteString('uitvoerbestand : ');
Terminal.ReadString(naam);
Terminal.WriteLn;
Create(uitvoer, naam, textMode, noReplace, status);
IF status # ok
THEN
  VerwerkStatus(status)
END;
SetOutput(uitvoer);

(* initieer de toestand en het huidige woord *)
toestand := nietInEenWoord;
woord := '';

(* verwerk het invoerbestand *)

(* lees het eerste teken *)
ReadChar(teken);

(* verwerk het teken *)
WHILE NOT EOT() DO

  (* verwerk de toestand 'einde regel' *)
  IF EOL()
  THEN
    ReadLn();
    teken := eol
  END;

  (* zet een hoofdletter om tot een kleine letter *)
  IF ('A' <= teken) AND (teken <= 'Z')
  THEN
    teken := CHR(ORD(teken) - ORD('A') + ORD('a'))
  END;
```

```
  (* verwerk het teken *)
  CASE toestand OF
    inEenWoord :
    IF ('a' <= teken) AND (teken <= 'z')
    THEN
      tekenStr[0] := teken;
      Concat(woord, tekenStr, woord, succes)
    ELSE
      WriteString(woord);
      WriteLn;
      IF status # ok
      THEN
        VerwerkStatus(status)
      END;
      toestand := nietInEenWoord
    END

  | nietInEenWoord :
    IF ('a' <= teken) AND (teken <= 'z')
    THEN
      tekenStr[0] := teken;
      Assign(tekenStr, woord, succes);
      toestand := inEenWoord
    END
  END;

  ReadChar(teken)
END;

(* sluit de bestanden af *)
Close(invoer, status);
Close(uitvoer, status);
IF status # ok
THEN
  VerwerkStatus(status)
END

END SplitsInWoorden.
```

19 Beheer van een inhoudstabel

19.1 Module Directory

De module Directory bevat bewerkingen voor het werken met bestanden op basis van de bestandsnaam in de plaats van een bestandsvariabele.

```
DEFINITION MODULE Directory;
FROM Files IMPORT FileState;

EXPORT QUALIFIED
  (* type *) FileNameType, DirQueryProc,
  (* proc *) Rename, Delete, DirQuery, TypeOfFileName;

TYPE FileNameType = (invalidName, singleName, wildName);
     DirQueryProc = PROCEDURE(ARRAY OF CHAR, VAR BOOLEAN);

PROCEDURE Rename  (    vanNaam  : ARRAY OF CHAR;
                       naarNaam : ARRAY OF CHAR;
                   VAR status   : FileState);

PROCEDURE Delete  (    naam     : ARRAY OF CHAR;
                   VAR status   : FileState);


PROCEDURE TypeOfFileName(naam   : ARRAY OF CHAR) : FileNameType;

PROCEDURE DirQuery(    naam     : ARRAY OF CHAR;
                       dirProc  : DirQueryProc;
                   VAR status   : FileState);
END Directory.
```

Het type FileNameType definieert drie eigenschappen voor bestandsnamen : invalidName betekent dat een bestandsidentificatie niet voldoet aan de syntaxis van het besturingssysteem; singleName is een ondubbelzinnige identificatie van een bestand; wildName is een bestandsidentificatie met jokers (Engels : wildcards) om een

hele groep bestanden aan te geven. Deze tekens zijn afhankelijk van het besturingssysteem. Dikwijls worden de symbolen '*', '?' '[' en ']' hiervoor gebruikt.

De procedure Rename wijzigt de oorspronkelijke bestandsnaam <vanNaam>, met de eigenschap singleName, in <naarNaam>. De procedure Delete wist de bestandsnaam <naam>, met de eigenschap singleName, uit de inhoudstabel. Deze procedures accepteren geen bestandsnamen met jokers. De functieprocedure TypeOfFileName levert het type van de bestandsnaam af.

Met de procedure DirQuery kunnen we de inhoudstabel opvragen voor een bestandsidentificatie met jokers. De actuele waarde voor de parameter <naam> kunnen we vervangen door een bestandsnaam met of zonder jokers, de actuele parameter voor <dirProc> is een procedure met type DirQueryProc. De procedure DirQuery doorloopt de inhoudstabel en roept de actuele procedurewaarde voor <dirProc> aan voor elke bestandsnaam die aan de bestandsidentificatie voldoet. De invoerparameter voor de procedurewaarde <dirProc> is een bestandsnaam, de uitvoerparameter is een waarde met type BOOLEAN. Bij de waarde FALSE stopt procedure DirQuery met het onderzoeken van de inhoudstabel. De waarde TRUE betekent verder zoeken in de inhoudstabel.

Voorbeeld :
We schrijven een programmamodule Dir voor het afdrukken van de bestandsnamen in een inhoudstabel. De invoer is een bestandsidentificatie met of zonder jokers.

```
MODULE Dir;
FROM SimpleIO  IMPORT ReadString, ReadLn, WriteString, WriteLn;
FROM Directory IMPORT DirQuery;
FROM Files     IMPORT FileState;

PROCEDURE DrukBestandsNaam(    naam : ARRAY OF CHAR;
                           VAR ok   : BOOLEAN);
BEGIN
WriteString(naam);
WriteLn;
ok := TRUE
END DrukBestandsNaam;

VAR status : FileState;
    naam   : ARRAY [0..13] OF CHAR;
BEGIN
(* lees de bestandsidentificatie *)
ReadString(naam); ReadLn;

(* onderzoek de inhoudstabel *)
DirQuery(naam, DrukBestandsNaam, status);
END Dir.
```

De procedures Rename en Delete aanvaarden uitsluitend een bestandsnaam met de eigenschap singleName. Met de procedure DirQuery kunnen we ook deze bewerkingen definiëren voor een groep bestanden. We programmeren bijvoorbeeld de programmamodule Wis voor het wissen van een groep bestanden.

```
MODULE Wis;
FROM SimpleIO  IMPORT ReadString, ReadLn;
FROM Directory IMPORT DirQuery, Delete;
FROM Files     IMPORT FileState;


PROCEDURE WisBestandsnaam(    naam   : ARRAY OF CHAR;
                          VAR succes : BOOLEAN);
VAR status : FileState;
BEGIN
Delete(naam, status);
succes := status = ok
END WisBestandsnaam;


VAR status : FileState;
    naam   : ARRAY [Ø..13] OF CHAR;
BEGIN
(* lees de bestandsidentificatie *)
ReadString(naam);
ReadLn;

(* onderzoek de inhoudstabel *)
DirQuery(naam, WisBestandsNaam, status);
END Dir.
```

20 De aanroep van subprogramma's

20.1 Module Program

De module Program verschaft de bewerkingen voor het aanroepen van programma's vanuit een actief programma, net zoals het aanroepen van de procedures in een Modula-2 programma. De verwerking van het aanroepende programma wordt onderbroken en het aangeroepen programma wordt uitgevoerd. Na de beëindiging van het aangeroepen programma wordt de verwerking van het aanroepende programma voortgezet. Bij talrijke implementaties blijven de modulen die het aanroepende en het aangeroepen programma gemeenschappelijk hebben, bewaard in het geheugen. Gegevens kunnen aldus tussen de programma's worden uitgewisseld.

```
DEFINITION MODULE Program;
EXPORT QUALIFIED
  (* type *) CallResult,
  (* proc *) Call, Terminate,
             SetInitialisation, SetTermination;
TYPE CallResult = (normalReturn, programHalt, keyboardHalt,
                   missingProgram, missingModule, duplicateModule,
                   versionError, codeError,
                   programCheck, ioError, otherError);


PROCEDURE Call     (    programmaNaam  : ARRAY OF CHAR;
                    VAR callResultaat  : CallResult);

PROCEDURE Terminate(  reden            : CallResult);

PROCEDURE SetInitialisation(initProc : PROC);

PROCEDURE SetTermination   (termProc : PROC);
END Program.
```

Door de procedure Call wordt een programma met de naam <programmaNaam> aangeroepen. De vorm van deze naam is afhankelijk van het besturingssysteem en bevat eventueel verwijzingen naar inhoudstabellen.

Het resultaat van de aanroep wordt geregistreerd in de parameter <callResultaat> met type CallResult. Het type CallResult beschrijft de mogelijke resultaatwaarden voor de aanroep van een programma :

normaal einde :

normalReturn : het aangeroepen programma is normaal afgelopen;

abnormaal einde :

programHalt : het aangeroepen programma is gestopt door de standaardprocedure HALT;

keyboardHalt : het aangeroepen programma is met de hand gestopt door de gebruiker;

laadfouten :

missingProgram : het laadprogramma heeft de hoofdprogrammamodule niet gevonden;

missingModule : het laadprogramma heeft een module niet gevonden;

duplicateModule : het laadprogramma ontdekt dubbelzinnige modulen;

versionError : het laadprogramma ontdekt inconsistente versienummers;

codeError : het laadprogramma ontdekt een codefout tijdens het laden van een module;

uitvoeringsfouten :

programCheck : uitvoeringsfout (bijvoorbeeld overloop, buiten bereik, te weinig geheugen);

ioError : in- of uitvoerfout;

andere fouten :

otherError

Met de procedure Terminate kan een aangeroepen programma vroegtijdig worden beëindigd. De actuele waarde voor de parameter <callResultaat> wordt doorgegeven aan het aanroepende programma. Als programma's met aanroepen worden geketend, kunnen we aldus het resultaat van een aanroep in de ketting propageren tot het programma stopt.

Voorbeeld :
In dit voorbeeld roept programma 'A' programma 'B' aan en programma 'B' roept programma 'C' aan. Als deze aanroep mislukt, wordt de verwerking van programma 'B' gestopt. Het resultaat van de aanroep van 'C' wordt door 'B' doorgegeven naar programma 'A'.

```
MODULE A;
FROM Program IMPORT Call, CallResult;
...
VAR resultaat : CallResult;
BEGIN

Call('B',resultaat);
IF resultaat # normalReturn
THEN
  ...
END
END A.

MODULE B;
FROM Program IMPORT Call, Terminate, CallResult;
...
VAR resultaat : CallResult;
BEGIN

Call('C',resultaat);
IF resultaat # normalReturn
THEN
  Terminate(resultaat)
END

END B.
```

Met de procedures SetInitialisation en SetTermination kunnen we procedures in de bibliotheekmodule installeren. Deze procedures worden juist voor en onmiddellijk na het aanroepen van een programma uitgevoerd. De procedure SetInitialisation installeert een of meer procedures met type PROC die automatisch voor het aanroepen van een subprogramma worden uitgevoerd. De procedure SetTermination installeert een of meer procedures die na het aanroepen van een programma worden uitgevoerd.

Voorbeeld :
We definiëren in dit voorbeeld een menugestuurd systeem. Op het beeldscherm worden de keuzemogelijkheden weergegeven. De gebruiker kiest een van deze mogelijkheden door het aanslaan van een functietoets. Een voorbeeld van een dergelijk menu is :

```
F1   Wijzigen van een tekstbestand
F2   Compileren van een module
F3   Koppelen van een programma
F4   Uitvlooien van de geheugendump
F5   Einde werksessie
```

Elk menu wordt bewaard op een extern tekstbestand. De structuur van dit bestand is als volgt : voor elke keuzemogelijkheid, behalve de laatste, bevat dit bestand twee regels. De eerste regel bevat de tekst die op het scherm wordt weergegeven, de tweede regel de naam van het overeenkomstige programma. De inhoud van dit bestand is dan bijvoorbeeld :

```
Wijzigen van een tekstbestand
mod
Compileren van een module
comp
Koppelen van het programma
link
Uitvlooien van een geheugendump
pmd
Einde werksessie
```

De namen van de programma's zijn afhankelijk van het besturingssysteem en van het Modula-2 systeem.
We definiëren de module Menu voor het verwerken en het gebruik van een dergelijk menubestand.

```
DEFINITION MODULE Menu;
FROM Toetsenbord IMPORT Functie;
EXPORT QUALIFIED
  (* type *) Menu,
  (* proc *) LeesMenu, TekenMenu, VoerMenuRegelUit, MenuEinde;

TYPE Menu = SET OF Functie;

PROCEDURE LeesMenu(naam : ARRAY OF CHAR);
PROCEDURE TekenMenu;
PROCEDURE VoerMenuRegelUit(keuze : Functie);
PROCEDURE MenuEinde(keuze : Functie) : BOOLEAN;
END Menu.
```

De procedure LeesMenu leest de gegevens voor een menu van een extern bestand met bestandsnaam <naam>. Deze gegevens worden bewaard in een verborgen gegevensstructuur, het huidig menu. De bewerkingen TekenMenu, VoerMenuRegelUit en MenuEinde gebruiken alle het huidige menu. De procedure TekenMenu geeft de keuzemogelijkheden op het beeldscherm weer. Elke keuzemogelijkheid wordt voorafgegaan door het symbool voor de desbetreffende functietoets. De procedure VoerMenuRegelUit zorgt voor het uitvoeren van een menuregel. De actuele waarde voor <keuze> is de waarde voor een functietoets met type Functie. Met de functieprocedure MenuEinde testen we de keuze voor het beëindigen van het menubesturingsprogramma. Dit besturingsprogramma heeft de volgende structuur :

```
VAR functie : Functie;

LOOP
  functie := FunctieToets();
  IF MenuEinde(functie)
  THEN
    EXIT
  END;
  VoerMenuRegelUit(functie)
END
```

Voor het aanroepen van een subprogramma voor het uitvoeren van een keuze wordt eerst het scherm gewist. We installeren hiervoor de procedure Terminal.EraseScreen als een initieerprocedure met de opdracht :

```
SetInitialisation(EraseScreen)
```

Na het verwerken van een keuze moet het menu opnieuw op het scherm worden getekend. We installeren de procedure TekenMenu als eindprocedure met de opdracht

```
SetTermination(TekenMenu)
```

De volledige oplossing voor het menubesturingsprogramma is dan :

```
MODULE BestuurMenu;
FROM Menu       IMPORT Menu, LeesMenu, TekenMenu, VoerMenuRegelUit,
                       MenuEinde;
FROM Program    IMPORT SetInitialisation, SetTermination;
FROM Terminal   IMPORT EraseScreen;
FROM ToetsenBord IMPORT Functie, FunctieToets;

VAR f : Functie;

BEGIN
SetInitialisation(EraseScreen);
SetTermination(TekenMenu);

LeesMenu('Modula-2');
TekenMenu;
LOOP
  f := FunctieToets();
  IF MenuEinde(f)
  THEN
    EXIT
  END;
  VoerMenuRegelUit(f)
END
END BestuurMenu.
```

We besluiten met de implementatie van de module Menu :

```
IMPLEMENTATION MODULE Menu;
FROM Text        IMPORT ReadString, ReadLn;
FROM Files       IMPORT File, FileState, ReadWriteMode, BinTextMode,
                        Open, Close, EOF;
FROM ToetsenBord  IMPORT GotoRowCol, WriteChar, WriteString,
                        EraseScreen;
FROM Program     IMPORT CallResult, Call, Terminate;

CONST keuzeAantal  = 10;
VAR   menu          : ARRAY [1..keuzeAantal] OF RECORD
        tekst     : ARRAY [0..60] OF CHAR;
        opdracht  : ARRAY [0..15] OF CHAR
      END;
      menuFuncties  : Menu;


PROCEDURE LeesMenu(n : ARRAY OF CHAR);
VAR f      : File;
    status : FileState;
    i      : CARDINAL;
```

```
BEGIN
(* initieer menu en menuFuncties *)
menuFuncties := Menu{};
FOR i := 1 TO keuzeAantal DO
  menu[i].tekst := '';
  menu[i].opdracht := ''
END;

(* lees het externe menubestand *)
Open(f, n, textMode, readOnly, status);
i := 1;
LOOP
  ReadString(f, menu[i].tekst, status);
  ReadLn(f, status);
  ReadString(f, menu[i].opdracht, status);
  IF EOF(f)
  THEN
    EXIT
  END;
  INCL(menuFuncties, VAL(Functie, i - 1));
  ReadLn(f, status);
  INC(i)
END;
Close(f, status)
END LeesMenu;


PROCEDURE TekenMenu;
CONST endStr = ØC;
VAR   i       : CARDINAL;

BEGIN
EraseScreen;
i := Ø;
REPEAT
  INC(i);
  GotoRowCol(4 + i, 2Ø);
  WriteChar('F');
  IF i < 1Ø
  THEN
    WriteChar(CHR(i + ORD('Ø')))
  ELSE
    WriteChar('1');
    WriteChar(CHR(i + ORD('Ø')))
  END;
  GotoRowCol(4 + i, 24);
  WriteString(menu[i].tekst)
UNTIL menu[i].opdracht[Ø] = endStr
END TekenMenu;
```

```
PROCEDURE VoerMenuRegelUit(n : Functie);
VAR res : CallResult;

BEGIN
Call(menu[ORD(n) + 1].opdracht, res);
IF res # normalReturn
THEN
  Terminate(res)
END
END VoerMenuRegelUit;

PROCEDURE MenuEinde(f : Functie) : BOOLEAN;
BEGIN
RETURN NOT (f IN menuFuncties)
END MenuEinde;

END Menu.
```

21 Het geheugenbeheer

21.1 De module Storage

De module Storage bevat de procedures voor het dynamisch beheer van het geheugen.

```
DEFINITION MODULE Storage;
FROM SYSTEM IMPORT ADDRESS, ADDRESSINC;

EXPORT QUALIFIED
  (* proc *) ALLOCATE, DEALLOCATE, CondAllocate;

PROCEDURE ALLOCATE     (VAR ptr     : ADDRESS;
                            grootte : ADDRESSINC);

PROCEDURE DEALLOCATE   (VAR ptr     : ADDRESS;
                            grootte : ADDRESSINC);

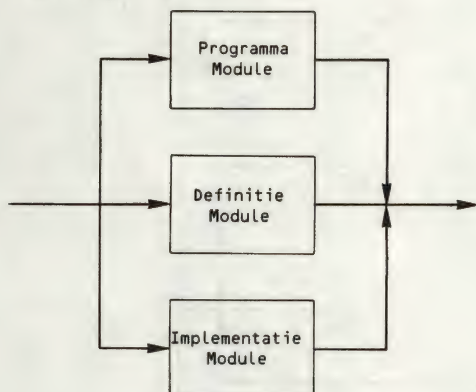
PROCEDURE CondAllocate (VAR ptr      : ADDRESS;
                            grootte  : ADDRESSINC;
                        VAR succes   : BOOLEAN);
END Storage.
```

De procedures ALLOCATE en DEALLOCATE worden door de compiler gebruikt voor de implementatie van de standaardprocedures NEW en DISPOSE. Als de toewijzing van een geheugenruimte met de procedure ALLOCATE mislukt, wordt het programma onderbroken. De procedure DEALLOCATE geeft de geheugenruimte weer vrij. De wijzervariabele <ptr> krijgt de waarde NIL.

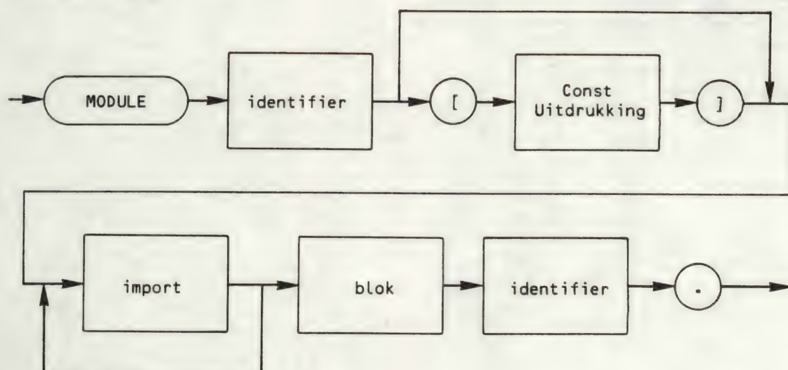
Met de procedure CondAllocate kan ook een geheugenruimte worden toegewezen. De waarde van de parameter <succes> is TRUE indien de toewijzing lukt. Anders is de waarde van <succes> gelijk aan FALSE en de waarde van <ptr> gelijk aan NIL. Met de procedure CondAllocate kunnen we aldus een tekort aan geheugenruimte in een programma opvangen.

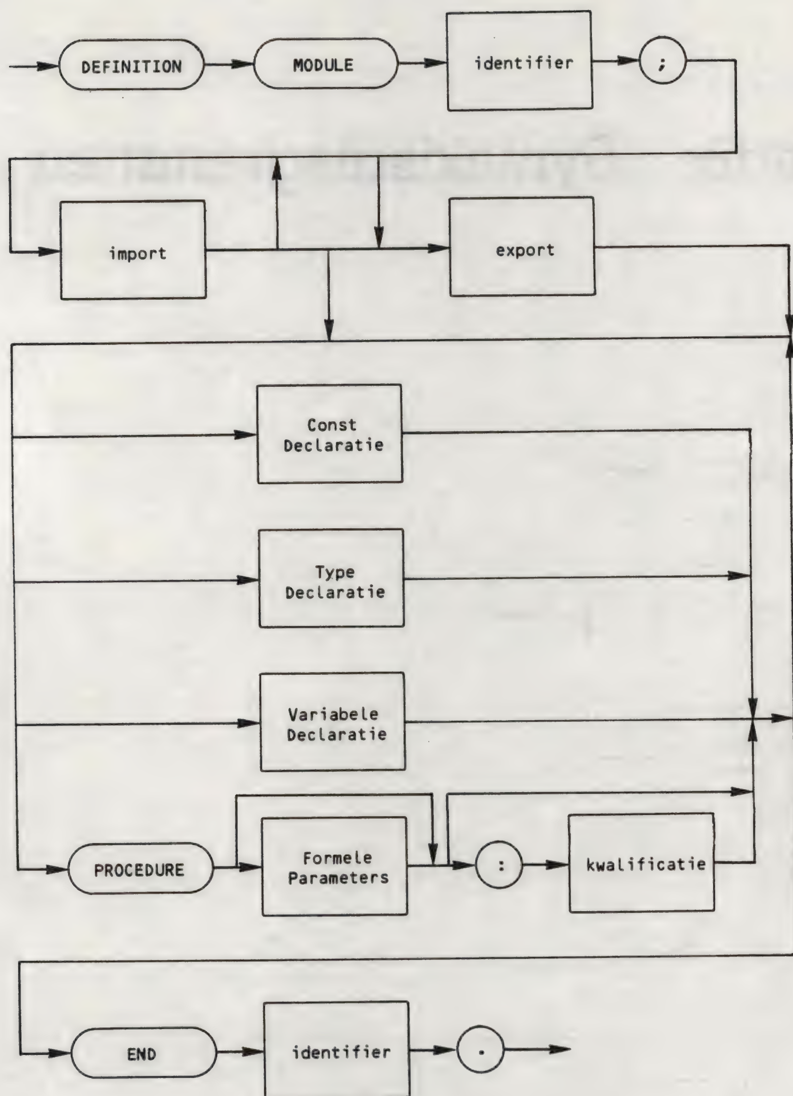
Appendix Syntaxisdiagrammen

CompileerEenheid

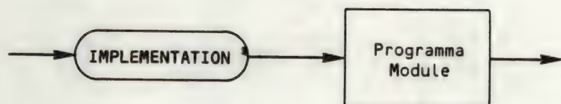


ProgrammaModule

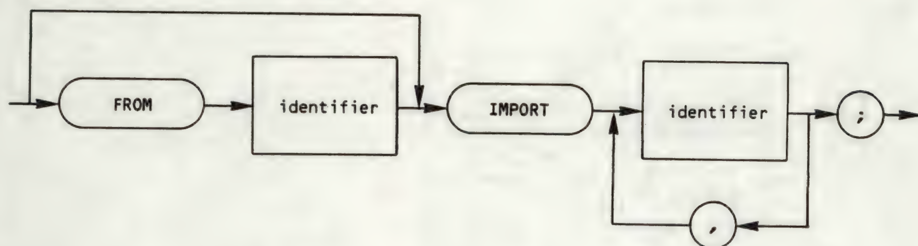



DEFINITION
MODULE
identifier
;
import
export
Const Declaratie
Type Declaratie
Variabele Declaratie
PROCEDURE
Formele Parameters
:
kwalificatie
END
identifier
.

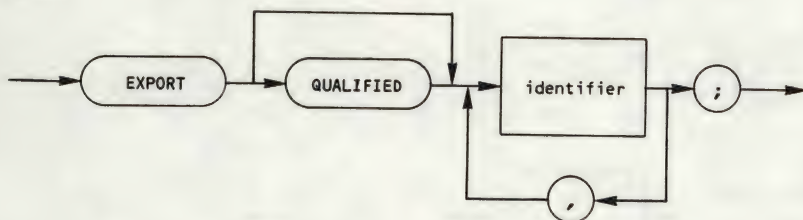
ImplementatieModule

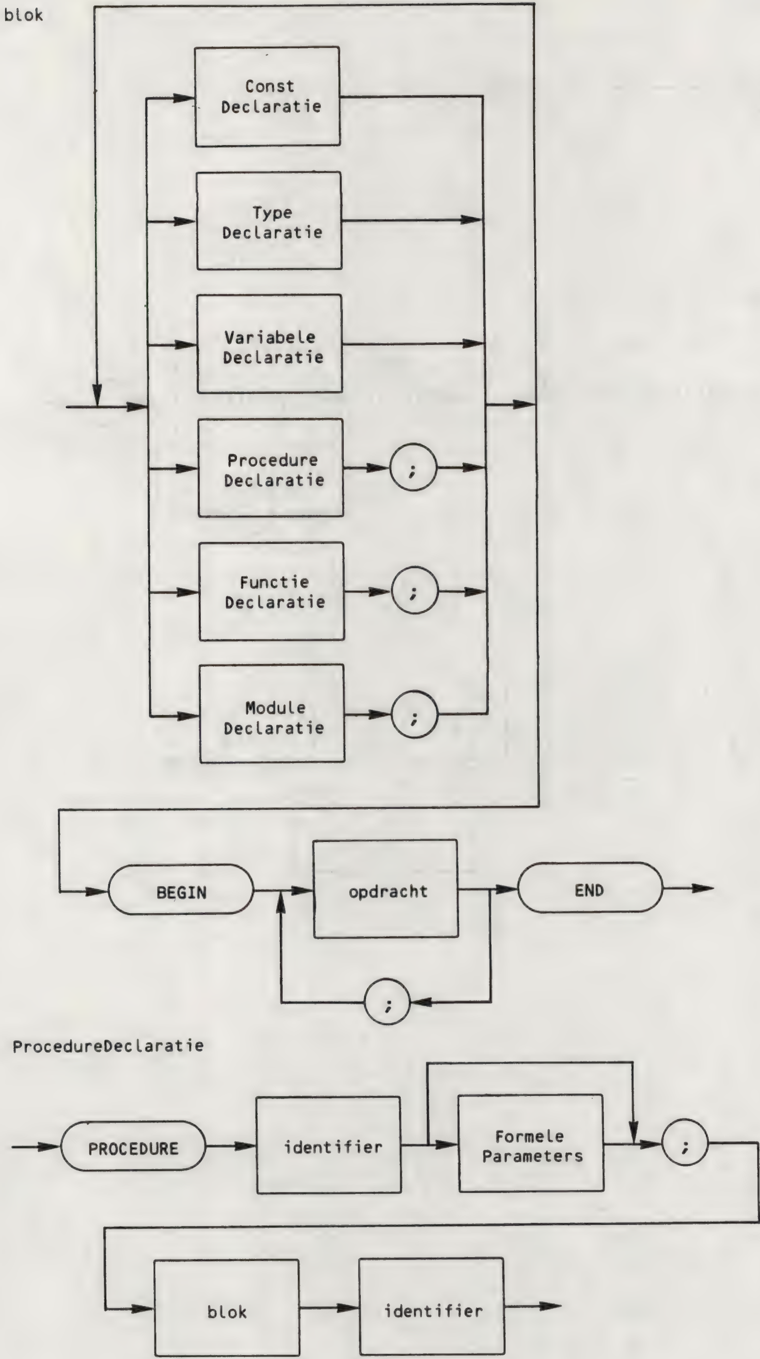


import

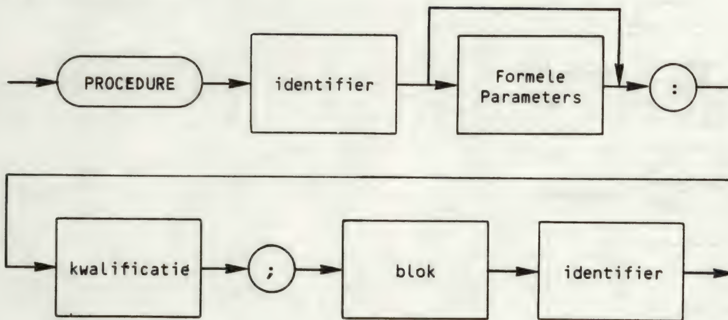


export

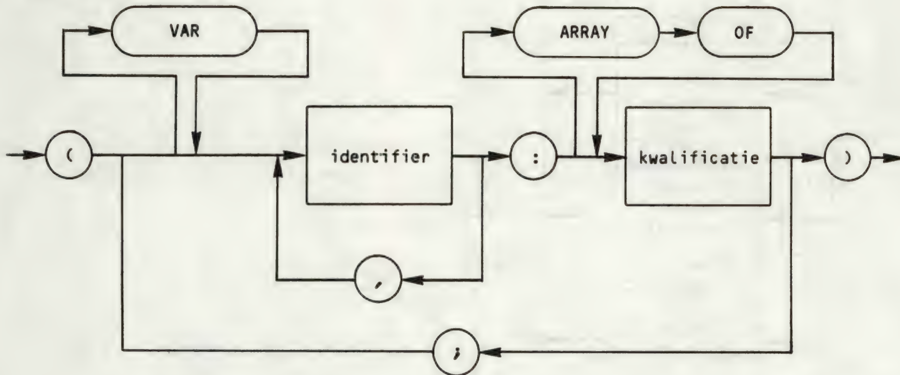



blok
Const Declaratie
Type Declaratie
Variabele Declaratie
Procedure Declaratie
;
Functie Declaratie
;
Module Declaratie
;
BEGIN
opdracht
END
;
ProcedureDeclaratie
PROCEDURE
identifier
Formele Parameters
;
blok
identifier

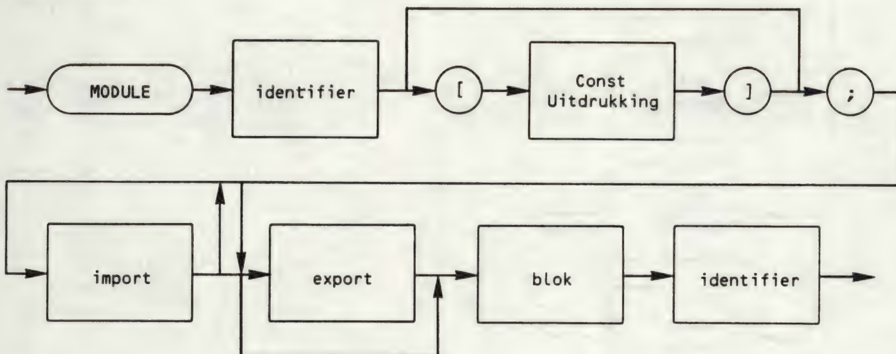
FunctieDeclaratie

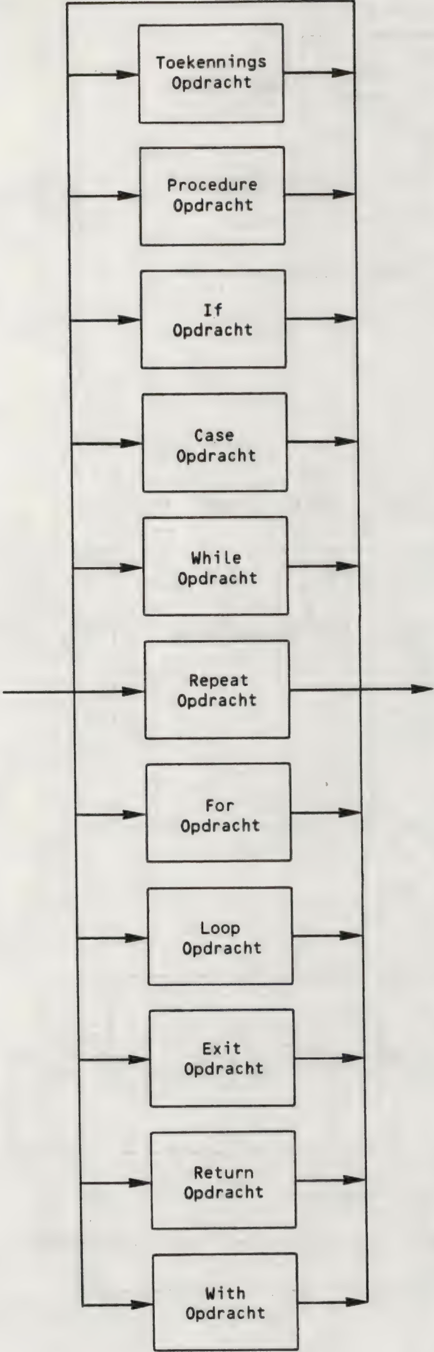


FormeleParameters

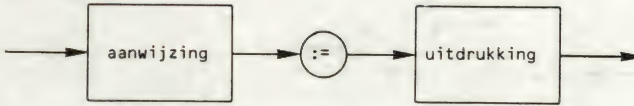


ModuleDeclaratie

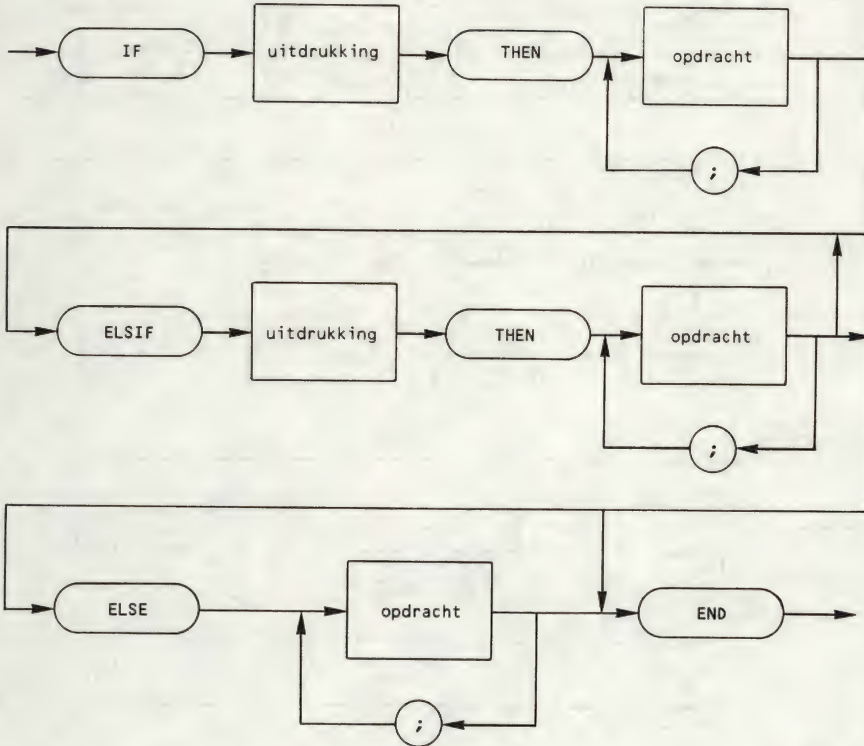



opdracht
Toekennings Opdracht
Procedure Opdracht
If Opdracht
Case Opdracht
While Opdracht
Repeat Opdracht
For Opdracht
Loop Opdracht
Exit Opdracht
Return Opdracht
With Opdracht

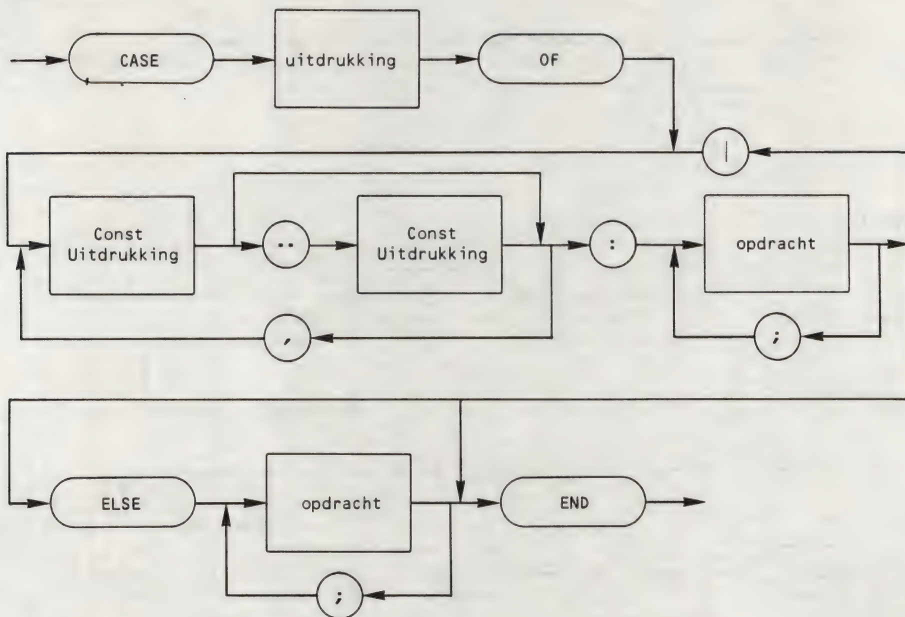
ToekenningsOpdracht



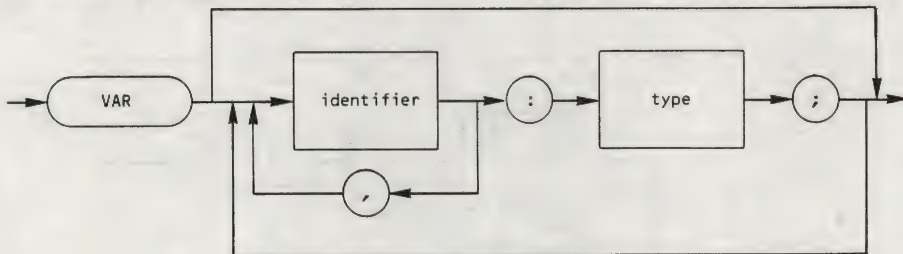
IfOpdracht



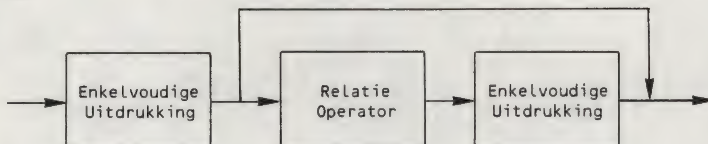
CaseOpdracht



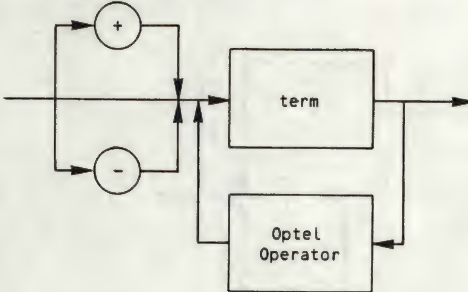
VariabeleDeclaratie



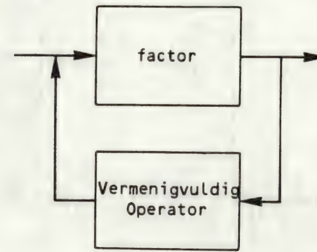
uitdrukking



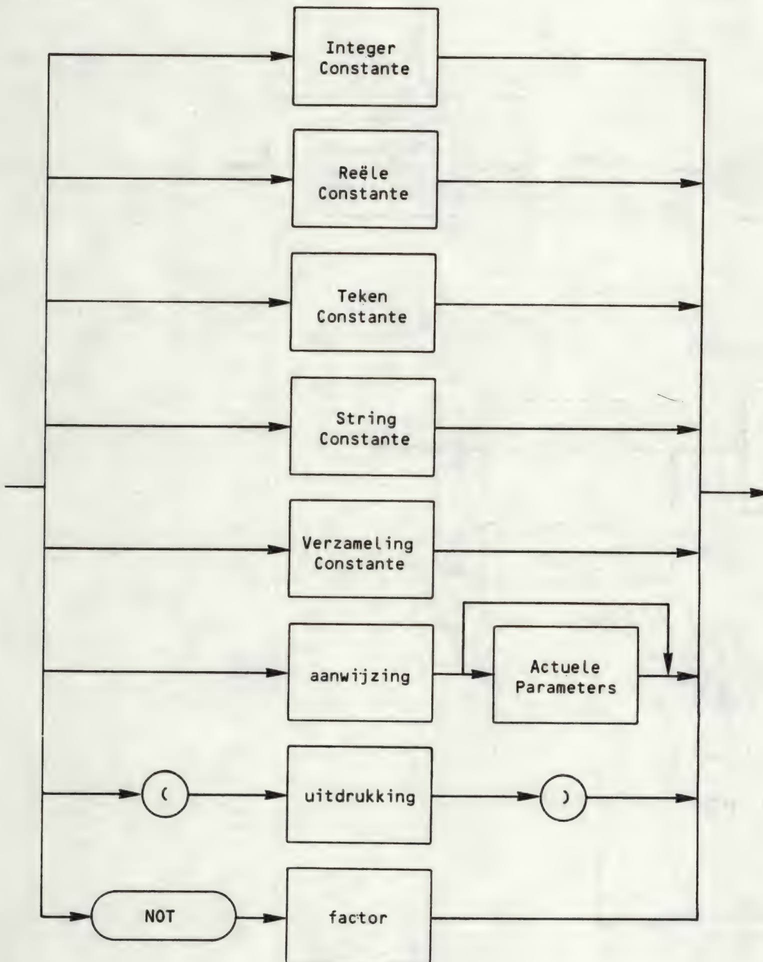
EnkelvoudigeUitdrukking



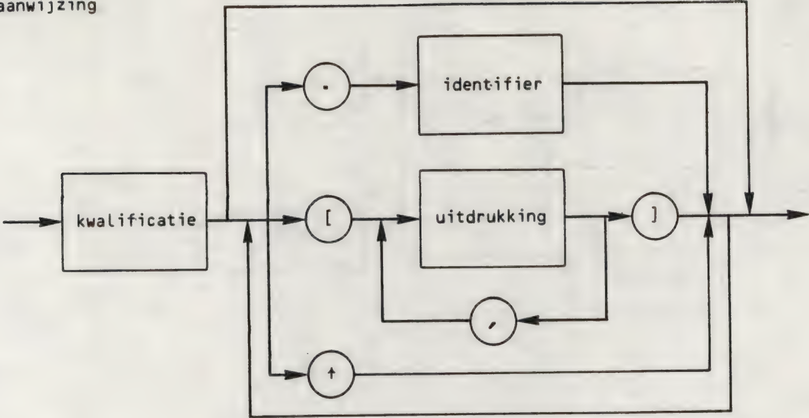
term



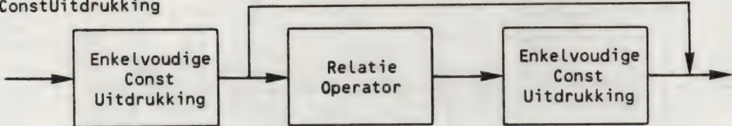
factor



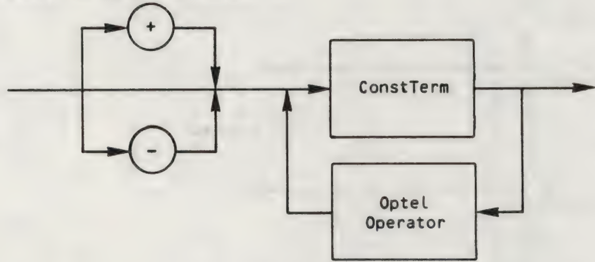
aanwijzing



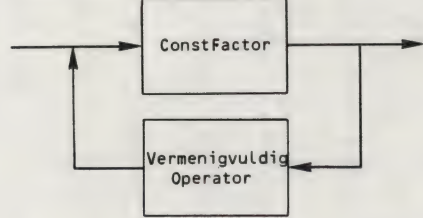
ConstUitdrukking



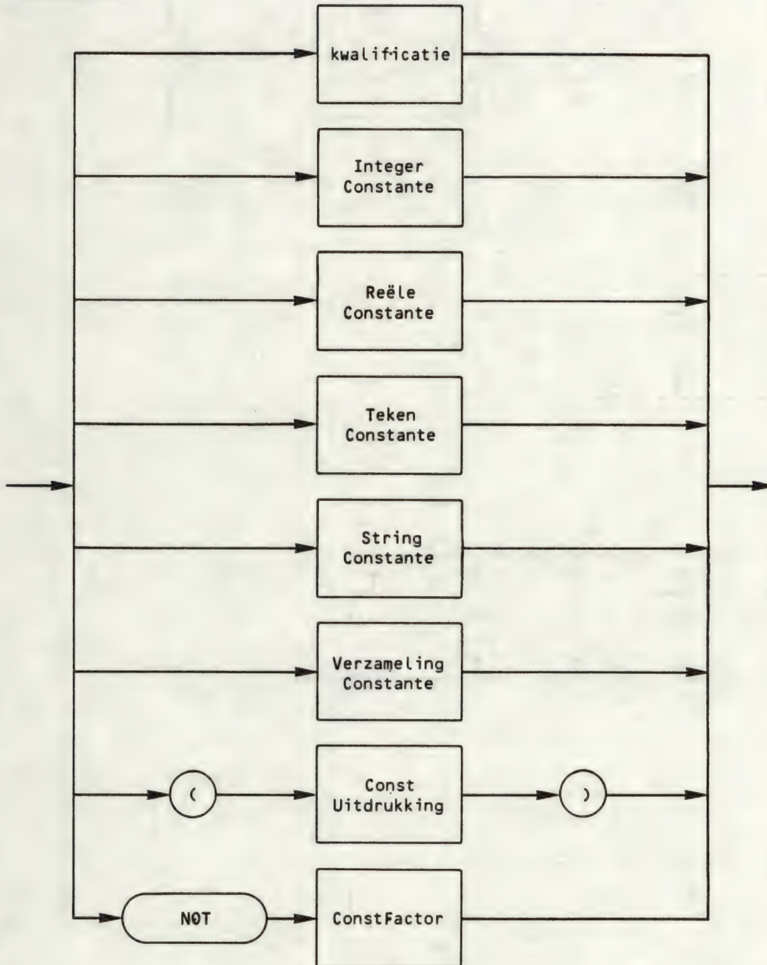
EnkelvoudigeConstUitdrukking



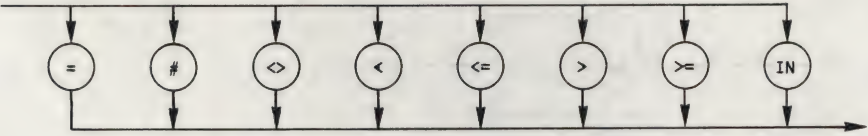
ConstTerm



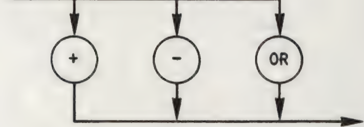
ConstFactor



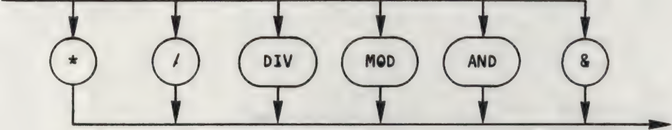
RelatieOperator



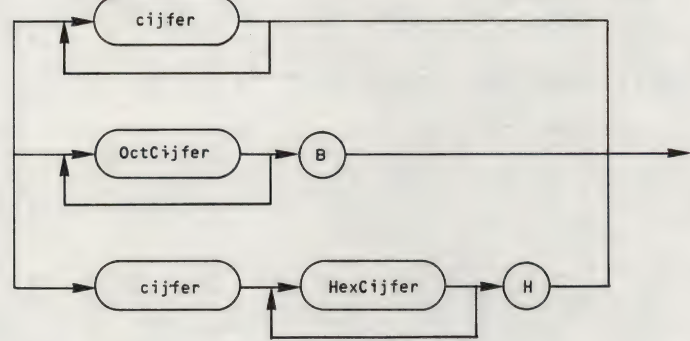
OptelOperator



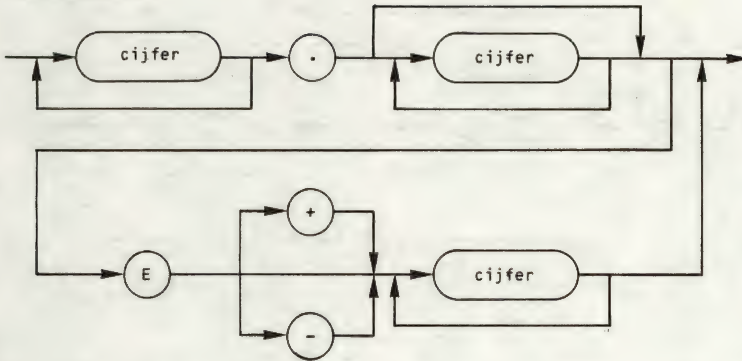
VermenigvuldigOperator



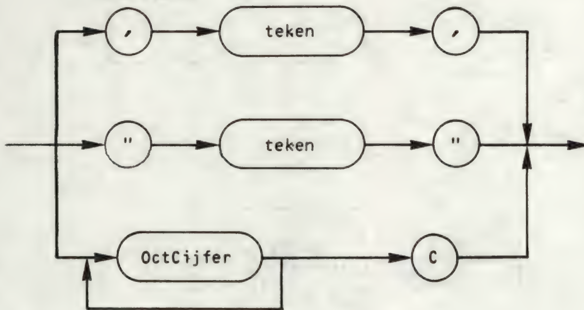
IntegerConstante



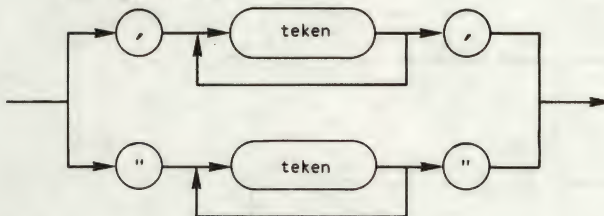
ReëleConstante



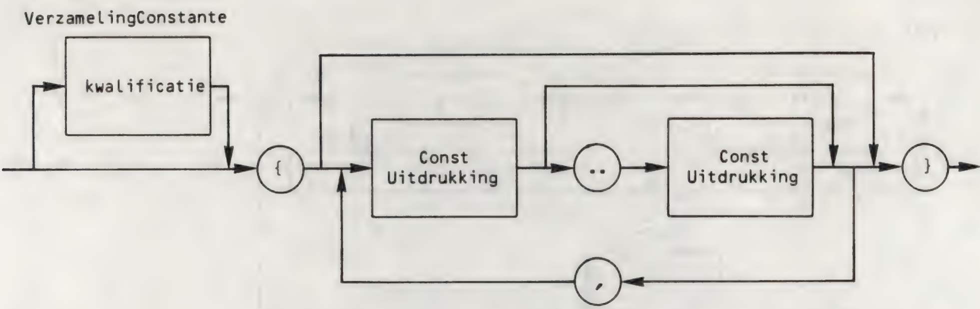
TekenConstante



StringConstante

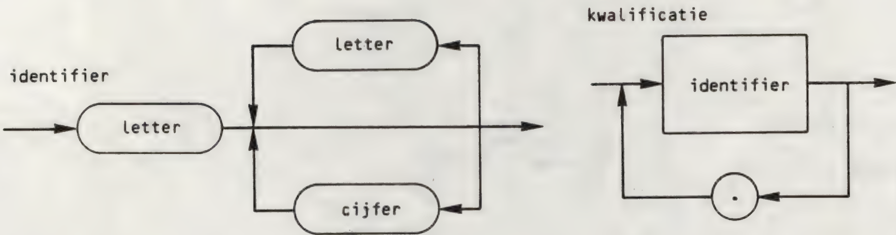


VerzamelingConstante

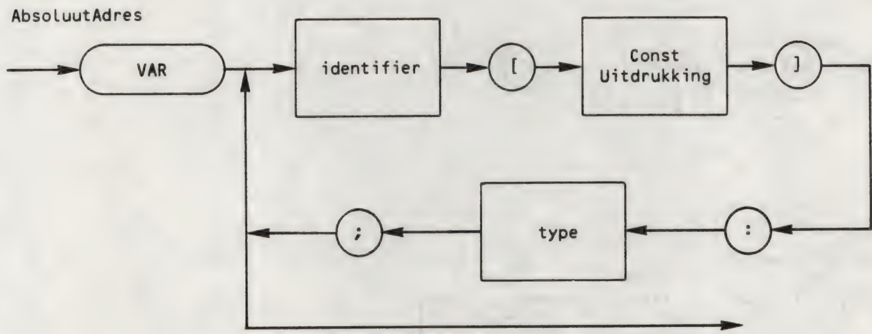


kwalificatie

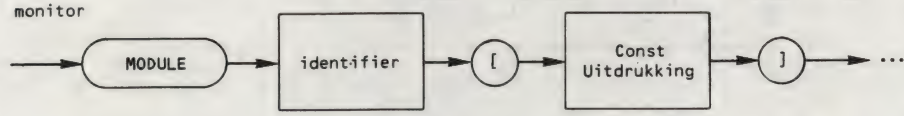
identifier



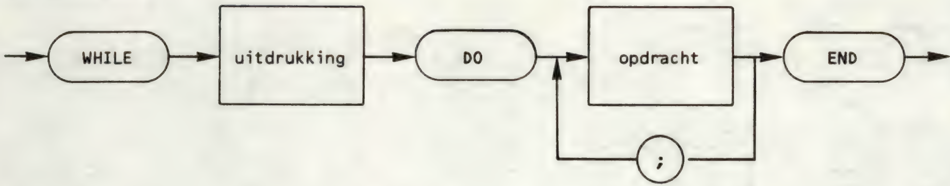
AbsoluutAdres



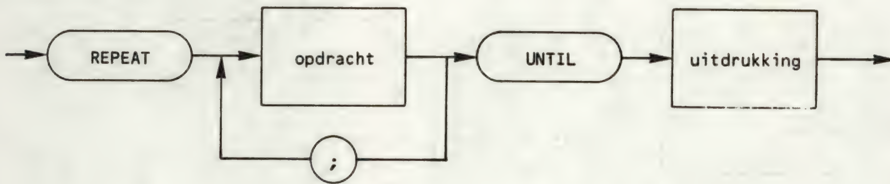
monitor



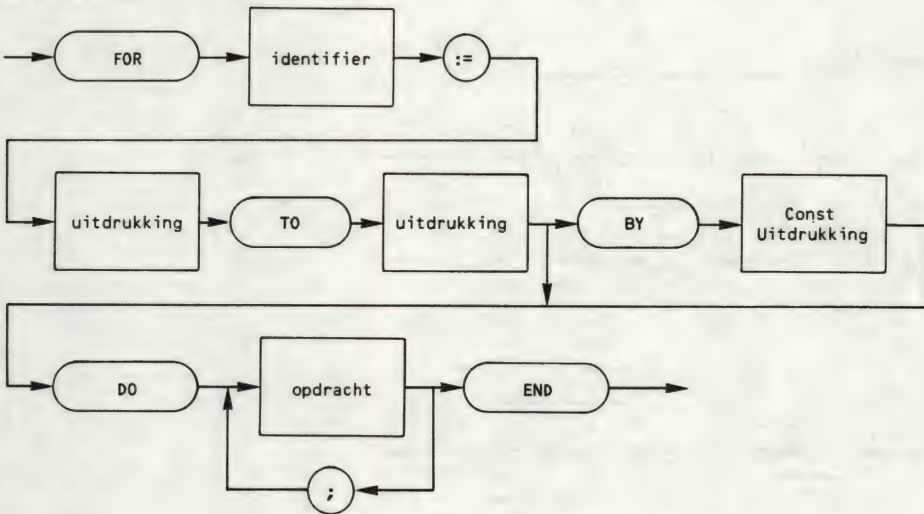
WhileOpdracht



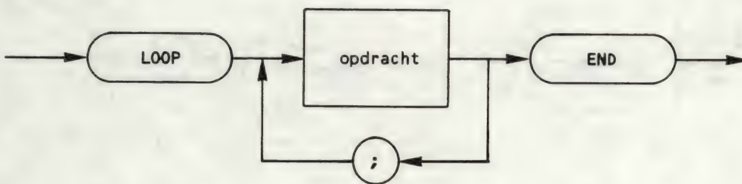
RepeatOpdracht



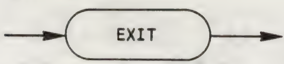
ForOpdracht



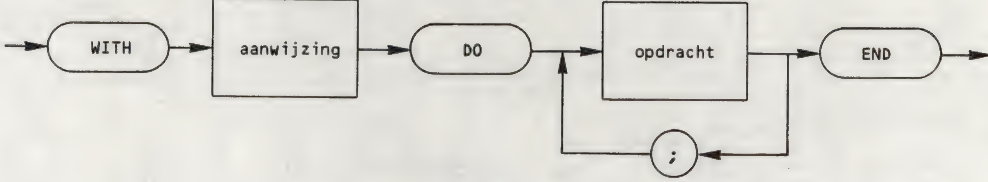
LoopOpdracht



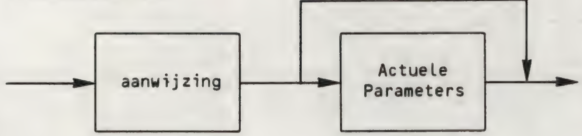
ExitOpdracht



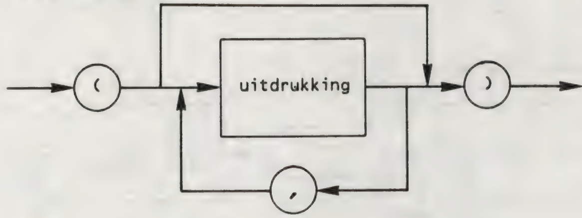
WithOpdracht



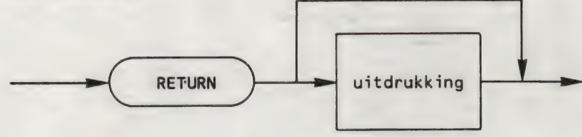
ProcedureAanroep



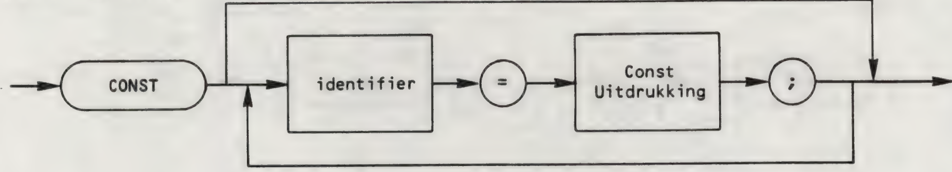
ActueleParameters



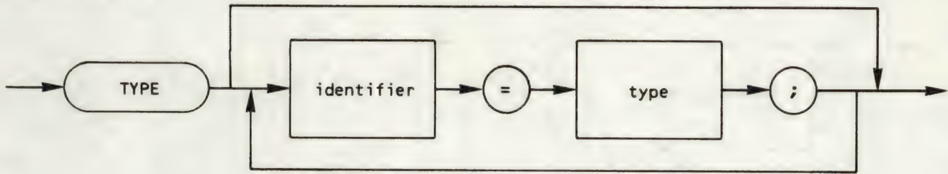
ReturnOpdracht



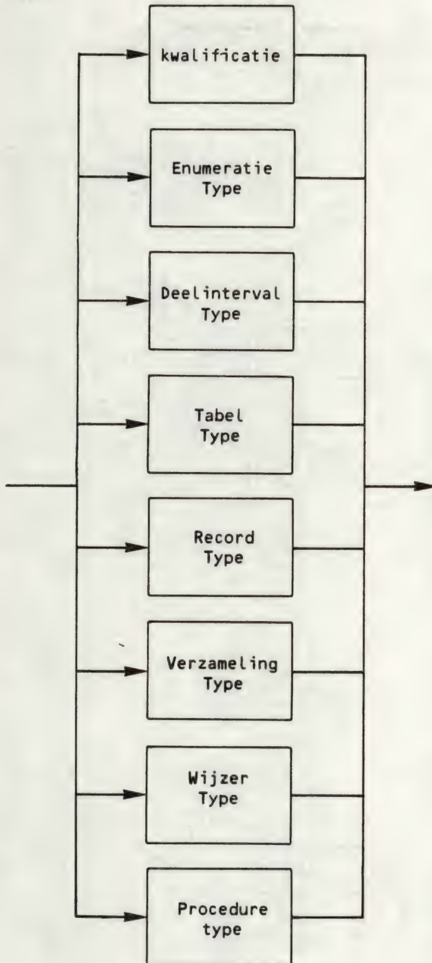
ConstDeclaratie



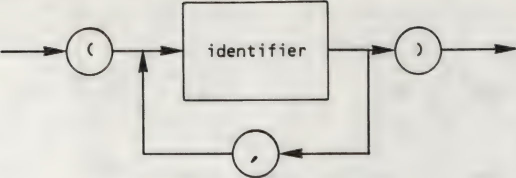
TypeDeclaratie



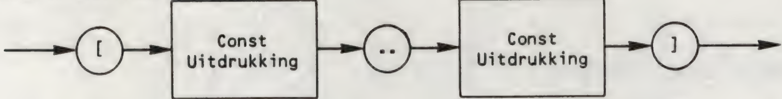
type



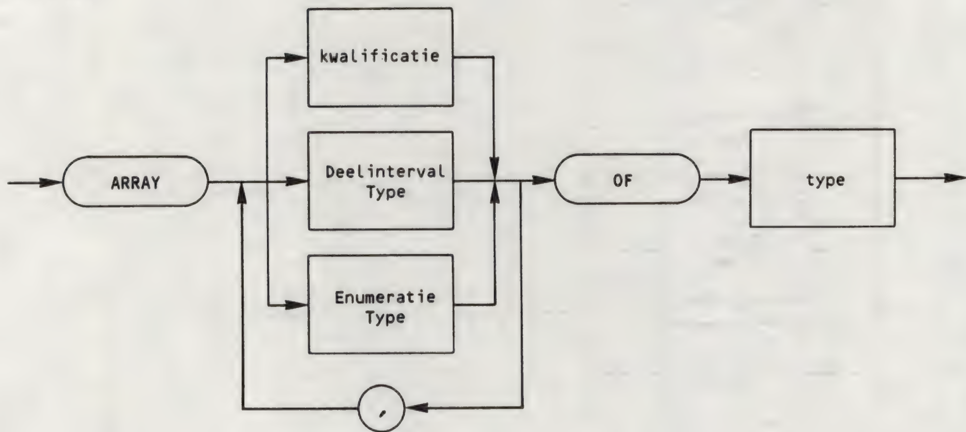
EnumeratieType



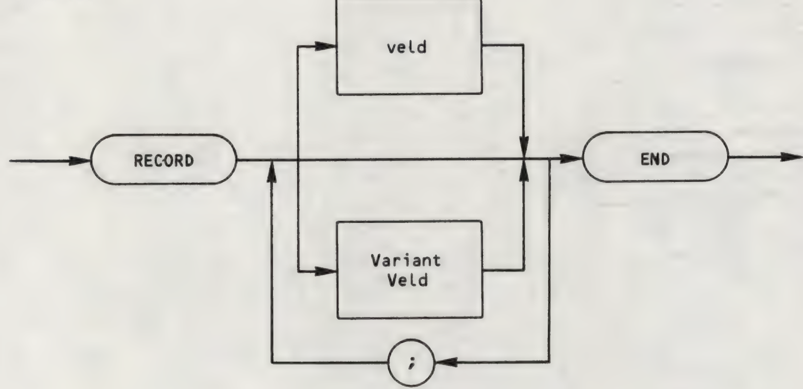
DeelintervalType



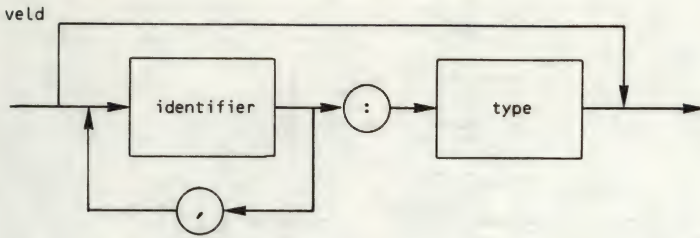
TabelType



RecordType



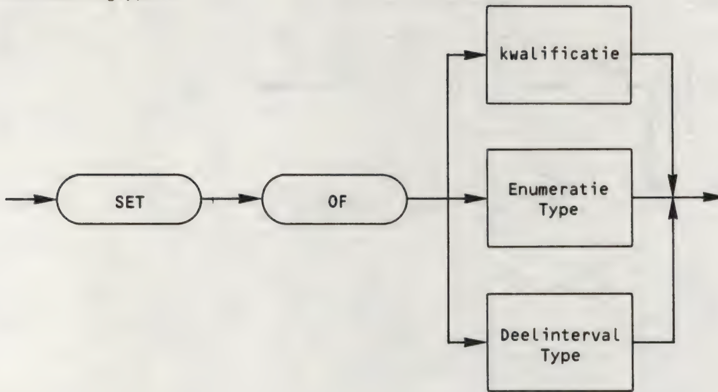
veld



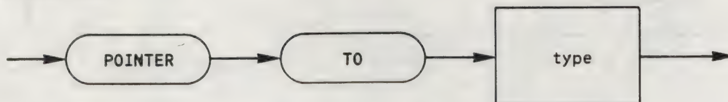
VariantVeld

CASE
identifier
:
kwalificatie
OF
Const Uitdrukking
..
Const Uitdrukking
,
:
veld
Variant Veld
;
|
ELSE
veld
Variant Veld
;
END

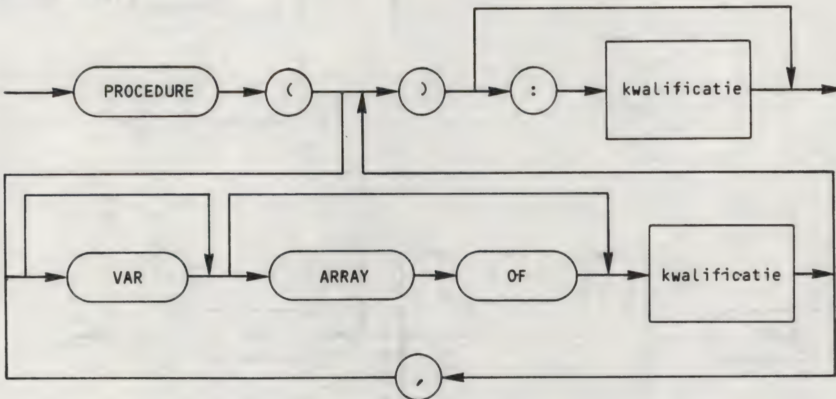
VerzamelingType



WijzerType



ProcedureType



Index

aandrijving 73
aanhalingstekens 42, 62
aanwijzing 124
abnormaal einde 31
ABS 50
abstract datatype 4, 16, 33, 240
abstractie 4, 209
abstractieschema 17
accessError 319
accolade 37
achtergrondkleur 311
actiedeel 81
actuele parameter 156, 161
ADA 1
ADDRESS 3, 195, 204
afsluiten van een bestand 321
aftrekken 49
afzonderlijke compilatie 4
ALLOCATE 22, 194, 224, 363
anagramprogramma 276, 278
AND 56
Arctan 303
ARRAY 3, 109
artikel 130
ASCII-verzameling 61
Assign 268, 269
atomair 36

basistype 132, 133
beeldscherm 186
beginconditie 92
begrenzingsteken 42, 43
bereik 51, 159, 194, 244-247
beschrijvende taal 35
bestand 5, 315
bestandsbeheer 305
bestandsidentificatie 169
bestandsnaam 26
bestandsvariabele 346
besturingsstructuur 81
besturingssysteem 24, 26, 39
besturingsteken 61, 266
besturingsvariabele 92, 94
bibliotheek 4, 24, 32, 211
bibliotheekmodule 10, 25, 169, 211

binair bestand 140, 141, 315
binair zoeken 115, 176
Binary 262, 316, 324
BinTextMode 140, 141, 316, 318
bitmap 138
bitmap-beeldscherm 1
BITSET 3, 137
bladzijde 127
blok 81, 105, 152, 153, 161
BOOLEAN 3, 56, 69, 76
bovengrens 73
breekpunt 31
bronbestand 26
BY 93
BY-clausule 94

C 33
CalcFilePos 326, 328
Call 355
CallResult 355
CAP 64
CARDINAL 3, 51, 76
CardToStr 295, 296
CASE 88
case-opdracht 2, 86
CaseOpdracht 88
CaseWaarde 88, 126
CaseWaardenlijst 88, 126
CHAR 3, 61, 76
CHR 62
Close 316, 321
codeError 356
commentaar 44
Compare 269, 274
CompareResult 268, 274
compilatie 29
compileereenheid 25, 212
compiler 24
compileroptie 27
complexiteit 4
Concat 269, 274
CondAllocate 363
conditie 59, 69, 83
conditiestructuur 83
conditionele if-opdracht 83
conditionele structuur 81, 83

CondRead 5, 305, 306, 331, 333, 343
consistentie 29, 232,
constante 49, 54
constantedeclaratie 64
constructie 24, 36
ConstUitdrukking 64
conversie 5, 53, 295
conversiefunctie 54, 76
Convert 262, 295
ConvertReal 262, 297
Cos 303
Create 316, 320
cross reference 24
cursorbesturing 308

datastructuur 4, 171
datavenster 32
DEALLOCATE 22, 194, 224, 363
debugger 31
DEC 51, 61, 67, 76
decimaal 32, 40
decimaalgetal 40
declaratie 2, 73, 161
declaratiedeel 81
declareren 64
deelinterval 73, 72
deelintervaltype 3, 67, 72, 76
deelprobleem 151
deelverzameling 136
def 26
definitiemodule 4, 16, 24-26, 29, 211, 217
Delete 268, 271, 351, 352
dereferentieoperator 192, 224
deviceError 319
Directory 262, 351
DirQuery 169, 170, 351, 352
DirQueryProc 169, 351
discreettype 76, 92
DISPOSE 193, 194
DIV 50
documentverwerking 127
doelbestand 26, 28
doorsnede 135
drijvende-komma-voorstelling 53
duplicateModule 356
dynamisch geheugenbeheer 5
dynamische matrix 198, 254
dynamische variabele 22, 191

EBNF 35
echo 345, 347
EchoMode 345, 347
editor 24
eindterm 36
eindwaarde 92
elektronisch werkblad 298
element 132
else-clausule 2, 89
empty statement 82
endError 319
endStr 265
enkelvoudig type 49
EnkelvoudigeUitdrukking 55, 56
EnkelvoudigType 72, 131
Entier 303
enumeratietype 67, 68, 72, 76 216, 253
Eof 316, 322, 331, 332, 343
Eol 5
Eoln 12
Eot 5, 343
equidistant 94
EraseScreen 186, 305, 307
EraseToEol 305, 307
EraseToEos 305, 308
etiketwaarde 197,
EXCL 136
existingFile 319
ExitOpdracht 97
Exp 303
export 211, 214, 244
Extended Backus-Naur Form 35
extraheren 267

factor 55, 57
faculteit 178
FALSE 56
fatale fout 315
Fibonacci 292
File 316
file-systeem 138
FilenameType 351
FilePosition 326, 327
FilePositions 262, 326
Files 262, 316, 331
Filestate 140, 141, 316
FLOAT 76
Flush 316, 323, 324
fonttype 309
FOR 90
for-opdracht 92
FormeelType 155, 158, 161
formele parameters 153, 155, 161, 162, 217, 263
FormeleTypeLijst 167
ForOpdracht 93
foutafhandeling 235, 237
foutboodschap 28
FPSectie 155, 161
FROM 214
from-clausule 244, 250, 251
functie 163
functie-identifier 53
functieprocedure 151, 161-165
functietoets 99, 308, 313
functionele abstractie 151

gebruik van bibliotheek-modulen 238

gebruikersmodule 4, 21, 31, 211, 213, 235
geheel getal 40
geheeltallig delen 49
geheugen 22
geheugenbeheer 363
geheugencel 195
geheugendump 24, 81
geheugeninhoud 32
geheugenruimte 22, 53, 154, 191, 224
geheugentoewijzing 21
gekwalificeerde identifier 215, 253
gereserveerde woorden 44
geschiedenis 1
gestructureerd programmeren 211
getal 39
getalrepresentatie 285
GetBof 326, 328
GetEchoMode 345
GetEof 326, 328
GetErrorInput 345
GetErrorOutput 345
GetFilename 316, 322, 324
GetFilePos 326, 328
GetInput 345, 346
GetLog 345
GetLogMode 345
GetOutput 345, 346
globale variabele 160, 232, 256
GotoRowCol 305, 307
grafisch uitvoerapparaat 138
grenswaarde 67
grondtal 296, 338
grootste gemene deler 8

HALT 32, 81
herhalingsopdracht 90
herhalingsstructuur 81, 174
hexadecimaal 32, 40
hexCijfer 40
hexGetal 40
HIGH 113, 118
hoofdletter 38
hulpmiddel 24

identifier 36-38, 49, 64
IdentLijst 67, 68
IfOpdracht 83, 84
implementatiemodule 16, 25, 211, 222
impliciete in- en uitvoer 216, 252
import 104, 211, 214, 244
IN 136, 137
INC 51, 61, 64, 72, 76
INCL 136
indirecte recursie 187
ingebouwde gegevensstructuur 239
inhoudstabel 5, 169, 170, 262, 351
initiëren 229
initiëring 21
Insert 268, 270
instellen van een bestand 321
integer 3, 49, 76
IntegerConstante 41
intensiteit 310
interactief 98, 99
interactieve debugger 31
interne module 211, 243
interne representatie 297
interval 51
IntToStr 295, 296
invalidName 351
invoerbestand 26
invoerlijst 10, 213
IOError 356
iteratie 174

joker 351

keyboardHalt 356
kleine letter 38
kleurenscherm 311
knooppunt 17, 199
kop 199
koppeling 157
koppelprogramma 24
kwalificatie 72, 134

laserprinter 127
lees-verwerk-strategie 102
leesbaarheid 2, 64, 160
leesopdracht 332
lege string 132
lege verzameling 132
lengte 42, 265
Length 268, 269
lettertype 309, 311
levensduur 160, 179, 191, 194, 232, 245, 247
lijst 198
Lilith 1
Lilith-implementatie 26
Ln 303
loader 24
logbestand 345
loggingOff 347
loggingOn 347
logische implicatie 60
logische uitdrukking 56, 83
LogMode 345, 347
lokale procedure 244
lokale variabele 179
LOOP 90
loop-opdracht 97
LoopOpdracht 97
lus 92

machinecode 140, 141
MathLib 262, 303
matixprodukt 121
matrix 117, 203
MAX 51, 52, 62, 76
meerdimensionaal 117
menugestuurd 99, 133, 358
meta-notatie 35
meta-symbool 36
microcomputer 33
MIN 51, 52, 62, 76
missingModule 356
missingProgram 356
MOD 26, 50
module 3, 33, 211
moduledeclaratie 243
moduledefinitie 3
modulekey 28
modulenaam 253, 254
modulesleutel 28, 29
modulevenster 32
muis 1

naamloze variabele 192
nameError 319
nesten van commentaar 45
neveneffect 165
NEW 193
niet-afdrukbaar 62
niet-eindterm 37
niet-fatale fout 315
noEcho 347
noMoreRoom 319
normalReturn 356
NOT 56
notOpen 319
null statement 82
NumberIO 263, 338
NumCols 305, 307
NumRows 305, 307
NumToStr 295, 296

octaal 40
octCijfer 40
octGetal 40
ODD 50
ok 318
omkeren 11
ondergrens 73
onderhoud 64
onderstrepen 310
ontwerp van bibliotheek-
  modulen 234
ontwikkelingsomgeving 24
opdracht 82, 103, 126, 161
opdrachtenrij 82
Open 316, 320
open-array-parameter 157,
  158, 201
opmaakstructuur 129, 130
optelOperator 55, 56
opzoeken 268

OR 56
ORD 62, 69, 76
otherError 356
outsideFile 319, 320
overdraagbaar 52
overdraagbaar type 74
overdraagbaarheid 261
overdrachtfunctie 62, 68
overloop 51

parameterbinding 154
parameterlijst 21, 213
Pascal 1, 8, 11, 33
personal computer 169
pijpleidingsprogramma 277
POINTER 3, 191
pop 11, 17, 20
Position 268, 272
post mortem debugger 31
Power 303
prettyprinter 24
prioriteitsregels 54
PROC 167
procedure 151
procedure-aanroep 156, 161
procedure-abstractie 16, 17
proceduredeclaratie 151, 160
procedurefunctie 66
procedurekop 151-153, 160
procedurenaam 39
procedureparameter 168, 227
proceduretype 166, 167, 235
procedurevariabele 166-169
procedurevenster 32
proces 5
produktieregel 36
produktiviteit 24
Program 263, 355
programCheck 356
programHalt 356
programma-object 49
programmamodule 10, 25, 26,
  103, 211, 212
programmeerhulpmiddel 24
programmeeronderwijs 33
programmeertaal 33
pseudo-recursief 176
puntkomma 82
Push 11, 17, 20

QUALIFIED clausule 244, 250,
  251

Read 12
ReadBlock 325, 326
ReadByte 325, 326
ReadBytes 325, 326
ReadCard 5, 339, 343
ReadChar 5, 305, 306, 331,
  332, 343
ReadInt 5, 339, 343
ReadLn 5, 331, 333, 343

ReadNum 5, 339, 343
ReadReal 344
ReadString 5, 276, 305, 331, 343
ReadWord 325
ReadWriteMode 140, 141, 316, 318
REAL 3, 53, 76
RealIO 344
RealToStr 297
rechte haak 37
record 3
recordtype 123, 216, 253
recursie 174,
redirection 305
ReëelGetal 41
ReëleConstante 41
ref 26
referentiebestand 26, 32
rekenkundige overloop 28
rekenkundige uitdrukking 54
rekenstring 285
relatie 136
relatieoperator 59, 60
relaties 136
Remove 316, 321
Rename 352
REPEAT 90
repeat-opdracht 95
RepeatOpdracht 95
ReplaceMode 316, 318
Reset 316, 322
resetState 316, 322
resolutie 138
rest 50
resultaattype 163
RETURN 81
ReturnOpdracht 163
Rewrite 316, 322
rij-per-rij-organisatie 203
runtime-systeem 24
ruwe venster 32

samengesteld type 109, 163
samenhangende procedures 239
samenvoegen 267
scalair 59, 163
schaalfactor 41, 53
schijfaandrijvingen 73
schrijfopdracht 334
scope 159
selectie 118, 124
selectiestructuur 83
selectieve if-opdracht 84
semantiek 35
sequentiële structuur 81, 82
SET 3, 131
SetEchoMode 345, 347
SetFilePos 326, 328
SetInitialisation 355, 357
SetInput 345, 346
SetLog 345
SetLogMode 345, 347
SetOutput 345, 346
SetTermination 355, 357, 359
SimpleIO 5, 20, 263, 343
Sin 303
singleName 351
software-ontwikkelaar 24
sorteren 139
specificatie 211
spoorwegprobleem 182
sprongadres 90
Sqrt 303
stack 11, 20
StackMod 17
standaardbibliotheek 261
standaardidentifier 39
standaardin- en -uitvoer 343
standaardinvoer 5, 345, 346
standaardomgeving 261
standaardtype 72
standaarduitvoer 345
StandardIO 345
stapeloverloop 28
State 316, 322
statische 191
statusovergangsdiagram 335
stijlvoorschrift 24
Storage 22, 194, 263, 363
string 5, 37, 42, 263-302
stringconstante 265,
stringvariabele 265, 266
StrToCard 295, 297
StrToInt 295, 297
StrToNum 295
subprogramma 355
Substring 268, 273
sym 26
symbolenbestand 232
symbolentabel 26, 28, 29
symbolische debugger 24, 31
symmetrisch verschil 135
syntactische factor 36
syntactische term 36
syntaxis 35
syntaxisdiagram 35, 37
syntaxisnotatie 35
systeemtaal 33
systeemtijd 249
SYSTEM 22, 195

tabeltype 109, 112, 117
techniek 24
tekenconstante 40, 62
tekstbestand 140, 141, 315, 331
tekstbesturingsteken 315
tekstscherm 305
tekstvenster 32
term 55, 57
Terminal 186, 263, 305
Terminate 355, 357

Text 263, 316, 331
tilde 56
TO-symbool 93
toegang tot een bestand 317
toekenningsopdracht 49, 52, 75, 119, 266
toestand van een bestand 318
toetsenbord 308, 313
torens van Hanoi 180
transparant 219
TRUE 56
TRUNC 76
Truncate 316, 322
TSIZE 22, 195
tussenvoegmethode 114
tweecomplement systeem 51
type 49, 64, 72
type-controle 29
type-overdrachtfunctie 52, 76
typedeclaratie 67, 72
TypeOfFilename 351, 352
typering 3

uitbreiding 26
UitdLijst 118
uitdrukking 60
uitvoerbestand 26
uitvoerlijst 213
UndoRead 5, 331, 333, 343

VAL 61, 69, 76
variabele 49, 66
variabele-parameter 154
variabeledeclaratie 67
variant 126, 196
VariantVeldLijst 126
veld 123
VeldenLijst 123
verborgen type 19, 22, 219
vereniging 135
vergelijken 268
vermenigvuldigen 49
vermenigvuldigoperator 55, 57
verschil 135
versieconflict 31
versiecontrole 29, 232
versiefouten 29, 30
versienummer 213, 232
versionError 356
vertolker 35
verwerk-lees-strategie 102
verzameling 132
verzamelingtype 131
volgorde bij compileren 59, 232
voorwaardelijke evaluatie 2
vrijgeven 21

waarde 49
waardeparameter 154
waarheidstabel 58
weergave van tekens 309
werkstation 1, 305
werkvoorschrift 49
wetgeving 35
WHILE 90
while-opdracht 90
WhileOpdracht 90
wijzertype 191
wijzervariabele 191
wildcards 351
wildName 351
willekeurig getal 246
Wirth 1
wiskundige functie 303
WITH 215
WithOpdracht 125
woordenlijst 334
woordenschat 37, 61, 261
woordlengte 51, 133
word 3
Write 12
WriteBlock 325, 326
WriteByte 325, 326
WriteBytes 325, 326
WriteCard 5, 339, 343
WriteChar 5, 305, 307, 331, 334, 343
WriteInt 5, 339, 343
WriteLn 5, 305, 307, 331, 343
WriteNum 5, 339, 343
WriteReal 344
WriteString 5, 275, 276, 305, 307, 331, 334, 343
WriteWord 325

Over de auteur

De auteur studeerde elektronica aan de Industriële Hogeschool te Antwerpen (IHAM) en wetenschappelijke computertoepassingen aan het Postuniversitair Centrum (PUC) te Hasselt. Hij is docent informatica in het Economisch Hoger Onderwijs. Hij was ook tijdelijk werkzaam als systeemontwerper bij een belangrijk Vlaams bedrijf voor kantoorautomatisering. Van de auteur verschenen eerder reeds het boek *Systeemprogrammatuur en Software-ontwikkeling voor Microcomputers* en populair-technische artikelen in verschillende Nederlandstalige tijdschriften.

Over het boek

Leren programmeren steunt op het begrijpen van elementaire concepten zoals data- en besturingsstructuren. Hiervoor gebruiken we bij voorkeur een notatie die de concepten duidelijk weergeeft en het gebruik van geordende structuren benadrukt. Met dit doel heeft Niklaus Wirth in 1968 de programmeertaal Pascal gedefinieerd.
Voor de ontwikkeling van softwareprojecten is Pascal echter minder geschikt. Talrijke Pascal-implementaties bevatten uitbreidingen om enkele tekortkomingen op te vangen, bijvoorbeeld de programmering van functies op laag niveau, de directe bestandsverwerking, stringmanipulaties. Deze uitbreidingen zijn sterk afhankelijk van de implementaties zodat van overdraagbaarheid van programma's nog nauwelijks kan worden gesproken. Ook bieden de uitbreidingen nog onvoldoende oplossingen voor moderne software-ontwikkeling.
Steunend op de principes van Pascal is in 1978 door Niklaus Wirth de nieuwe taal Modula-2 gedefinieerd. De belangrijkste eigenschap van deze taal is de systematische ondersteuning van de programmering met modulen. Met behulp van deze modulen kunnen we softwaresystemen ontwikkelen waarbij we gebruikmaken van moderne inzichten in programmering: abstracte datatypen, verbergen van informatie, scheiden van belangen.
In dit boek wordt in de eerste plaats de programmeertaal Modula-2 in detail behandeld. Ook wordt er veel aandacht geschonken aan moderne software-ontwikkeling. Hiervoor gebruiken we een als standaard voorgestelde modulebibliotheek. Het boek kan dienen als handboek bij een cursus over Modula-2 en over het ontwerpen van programmatuursystemen. Van de lezer wordt enige voorkennis gevraagd van Pascal en van het ontwerpen van algoritmen.

ISBN 90 6233 212 9